NORTHWESTERN UNIVERSITY

Variants of MaxRS Queries for Trajectories, Shapes and Spatial Data Streams

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Muhammed Mas-ud Hussain

EVANSTON, ILLINOIS

December 2018

ABSTRACT

Variants of MaxRS Queries for Trajectories, Shapes and Spatial Data Streams

Muhammed Mas-ud Hussain

We address the problem of efficient maintenance of the answer to a new type of query: Continuous Maximizing Range-Sum (Co-MaxRS) for moving objects trajectories. The traditional static/spatial MaxRS problem finds a location for placing the centroid of a given (axes-parallel) rectangle R so that the sum of the weights of the point-objects from a given set O inside the interior of R is maximized. However, moving objects continuously change their locations over time, so the MaxRS solution for a particular time instant need not be a solution at another time instant. In this dissertation work, we devise the conditions under which a particular MaxRS solution may cease to be valid and a new optimal location for the query-rectangle R is needed. More specifically, we solve the problem of maintaining the trajectory of the centroid of R. In addition, we propose efficient pruning strategies (and corresponding data structures) to speed-up the process of maintaining the accuracy of the Co-MaxRS solution. As in many real-world applications, trajectories data sets need to reside on a secondary storage or even on cloud, a sequential access to the whole trajectories is inefficient. To resolve this, we devise a

hierarchical grid-based index structure which can be built on-the-fly and maintained via KDS. Moreover, there are many scenarios where a fast response to the query is necessary, and an approximate solution that is closer to the current MaxRS solution is preferred. Thus, we present a novel approximation algorithm (with a bound on the approximation ratio) for the Co-MaxRS problem using the grid-based partitioning system. We prove the correctness of our approach and present experimental evaluations over both real and synthetic datasets, demonstrating the benefits of the proposed methods.

Moreover, we also address the problem of maintaining the correct answer-sets to the *Conditional Maximizing Range-Sum* (C-MaxRS) query in spatial data streams. In many practical settings, the objects from a particular set – e.g., restaurants – can be of different types – e.g., fast-food, Asian, etc. The C-MaxRS problem deals with maximizing the overall sum – however, it also incorporates class-based constraints, i.e., placement of r such that a lower bound on the count/weighted-sum of objects of interests from particular classes is ensured. We first propose an efficient algorithm to handle the static C-MaxRS query and then extend the solution in an event-based manner to handle dynamic (data streams) settings. Subsequently, we turn our attention to the case of bursty streaming inputs, which is common in many applications, – e.g., thousands of users go online (or, offline) at (or, nearly) the same time in large social networks such as Facebook. We show that dealing with events one by one is not efficient when processing bursty inputs and present a novel technique to cater to such scenarios, by creating an index over the bursty data on-the-fly and processing the collection of events in an aggregate manner. Our experiments over datasets of up to 100,000 objects show that the proposed solutions provide significant efficiency benefits over the naïve approaches.

Finally, we investigate another novel variant of the well-known MaxRS (Maximizing Range Sum) problem – namely, the MAxRS^3 (Maximizing Area-Range Sum for Spatial Shapes). As we've already mentioned, the MaxRS problem amounts to detecting a location where a fixed-size rectangle R should be placed, so that it covers a maximum number of points – or sum of weights, if the points are weighted – from a given input set of 2D points. While variants have tackled the settings in which the input set to MaxRS problem consists of polygons instead of points – the solution is still based on (weighted) count. We postulate that in many practical applications it is of interest to determine where to place the input rectangle so that the total area-coverage in its interior is maximized. In this work, we formalize the MAxRS^3 problem and propose (to our knowledge) the first solution to this new problem.

Acknowledgements

I take this opportunity to extend my sincere gratitude and appreciation to all those who made this Ph.D. thesis possible.

Foremost, I would like to express my heartfelt appreciation to my advisor Prof. Goce Trajcevski for the continuous support of my Ph.D study and research, for his patience, motivation, enthusiasm, and immense knowledge. Since my first day at Northwestern, prof. Trajcevski believed in me like nobody else and gave me endless support during the tough times. Whenever I lost my way, he was always there to put me right back on track. On the academic level, Prof. Trajcevski taught me fundamentals of conducting scientific research, how to work through a problem, and the art of academic writing. I am indebted to him for all my accomplishments during my graduate studies, and could not have imagined having a better advisor and mentor for my Ph.D study.

I would like to express my deepest gratitude to my family and friends, for their warm love, continued patience, and unconditional support. I am grateful to my ever-supporting mother and father, who have provided me through moral and emotional support in my life. I cannot stress enough the support my wife Bony provided during the difficult times we had to endure together to make all this possible. This dissertation would not have been possible without her love, and sacrifices. I should specially thank my eternal cheerleaders – niece Ayaat and elder brother Zico. I am also thankful to my mother-in-law and father-in-law, eldest brother Pele, sister-in-laws (Nafi and Mita), kids (Ayaan and Arnaaz), and friends (special shoutout to Ragib Ahsan and Dipendra Jha) who have supported me along the way.

Besides my advisor, I would like to thank the rest of my dissertation committee members (Prof. Peter Scheuermann, Prof. Aleksandar Kuzmanovic, and Dr. Bo Xu) for their great support and invaluable advice. I am thankful to Prof. Scheuermann and Prof. Kuzmanovic, experts of Computer Systems research, for their crucial remarks that shaped my final dissertation. I am also grateful to Bo, for his insightful comments and for sharing with me his tremendous experience in the spatial data management field – both during my internship at HERE and later, in exam presentations.

I would like to thank my lab mates and other academic collaborators for their continued support in various research projects. I'd specially mention my ex-labmates Besim Avci, Panitan Wongse-ammat (Top), and Bing Zhang – with whom I had worked on a number of successful projects.I would like to thank my undergrad thesis supervisor Prof. Mohammed Eunus Ali and his students (Ashik, Farabi, Naved, etc.) for their continued contribution and help. I spent several quarters as a research intern at three great companies – IBM Research, HERE, and Facebook, where I had the opportunity to work with fantastic researchers and mentors (Bo Xu, Matthew Davis, Linbin Yu, Matei Stroila, Eytan Bakshy – to name a few), and would like to thank them for their significant impact in my growth as a Computer Scientist.

Thanks are also due to the National Science Foundation (NSF) (Grants III 1213038, CNS 1646107, and CNS-0910952), Office of Naval Research (ONR) (grant N00014-14-10215), HERE (grant 30046005) and Northwestern University for their financial support that I otherwise would not have been able to undertake my research endeavors.

Dedication

This dissertation is dedicated to my wife, Bony Anjabeen, for her continued sacrifices and remarkable perseverance to make all of this possible.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

In this chapter, we first present the motivation and applications for Continuous MaxRS in moving objects databases, and then proceed with laying out the motivation for Conditional MaxRS for spatial data streams. Subsequently, we introduce another MaxRS variant, namely, Maximizing Area-Range Sum for Spatial Shapes (MAxRS^3). Finally, the basic technical background is laid out in Section 1.4.

1.1. Co-MaxRS

Recent technological advances in miniaturization of position-aware devices equipped with various sensors, along with the advances in networking and communications, have enabled a generation of large quantities of *(location, time)* data – O(Exabyte) [**50**]. This, in turn, promoted various geo-social applications where the *(location, time)* information is enriched with (sensed) values from multiple contexts [**76, 80**]. At the core of many such

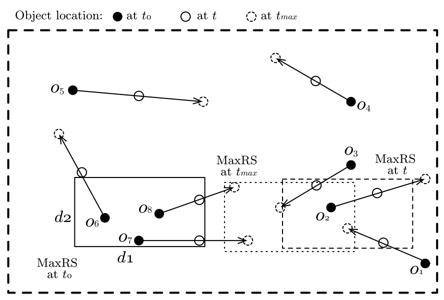


Figure 1.1. MaxRS vs. Co-MaxRS.

applications of high societal relevance – e.g., tracking in ecology and environmental monitoring, traffic management, online/targeted marketing, etc. – is the efficient management of mobility data [**62**].

Researchers in the Spatio-temporal [**47**] and Moving Objects Databases (MOD) [**27**] communities have developed a plethora of methods for efficient storage and retrieval of the whereabouts-in-time data, and efficient processing of various queries of interest. Many of those queries – e.g., range, (k) nearest neighbor, reverse nearest-neighbor, skyline, etc. – have had their "predecessors" in traditional relational database settings, as well as in spatial databases [**70**]. However, due to the motion, their spatio-temporal variants became continuous (i.e., the answer-sets change over time) and even persistent (i.e., answers change over time, but also depend on the history of the motion) [**54, 84**].

1.1.1. Motivation and Challenges

In a similar spirit, this work explores the spatio-temporal extension of a particular type of a spatial query – the, so called, Maximizing Range-Sum query (MaxRS), which can be described as follows:

Q: *"Given a collection of weighted spatial point-objects O and a rectangle R with fixed dimensions, finds the location(s) of R that maximizes the sum of the weights of the objects in R's interior".*

Various aspects of MaxRS (e.g., scalability, approximate solutions, insertion/removal of points) have been addressed in spatial settings [**17, 24, 38, 58, 64, 73**] – however, our main motivation is based on the observation that there are many application scenarios for which efficient processing of the *continuous* variant of MaxRS is essential. Consider the following query:

Q1: *"What should be the trajectory of a drone which ensures that the number of mobile objects in the Field-of-View of its camera is always maximal?"*.

It is not hard to adapt **Q1** to other application settings: – environmental tracking (e.g., optimizing a range-bounded continuous monitoring of a herd of animals with highest density inside the region); – traffic monitoring (e.g., detecting ranges with densest traffic between noon and 6PM); – video-games (e.g., determining a position of maximal coverage in dynamic scenarios involving change of locations of players/tanks in World of Tanks game). Pretty much any domain involving continuous detection of "most interesting" regions involving mobile entities is likely to benefit from the efficient processing of variants of **Q1** (e.g., mining popular trajectories patterns [**87**], sports analytics [**67**], etc.).

Contrary to the traditional *range* query which detects the number of points, or higher dimensionality objects such as (poly)lines and shapes, related to a given *fixed* region, the MaxRS determines the location for placing a given region so that the sum of the weights (i.e., some objective function related to location) is maximized. Originally, the MaxRS problem was tackled by the researchers in computational geometry [**38, 58**] – however, motivated by its importance in LBS-applications – e.g., best location for a new franchise store with a limited delivery range, most attractive place for a tourist with a restricted reachability bound – recent works have proposed scalable efficient solution for MaxRS in spatial databases [**17**], including approximate solutions [**73**] and scenarios where the weights may change and points may be added/deleted [**24**].

However, the existing solutions to MaxRS queries can only be applied to a specific time instant – whereas **Q1** is a *Continuous MaxRS* (Co-MaxRS) variant. Its weighted-version would correspond to prioritizing certain kinds of mobile objects (e.g., areas with most trucks – by assigning higher weights to trucks) to be tracked by the drone, or certain kinds of tanks in the World of Tanks game. The fundamental difference between MaxRS and Co-MaxRS is illustrated in Figure 1.1. Assuming that the 8 objects are static at time t_0 and the weights of all the objects are uniform, the placement of the rectangle R indicated in solid line is the solution, i.e., count for optimal R is 3. Other suboptimal placements are possible too at t_0, e.g., covering only o_2 and o_3 with count being 2. However, when objects are mobile, the placement of R at different time instants may need to be changed – as shown in Figure 1.1 for t_0, t and t_{max}.

1.1.2. Contributions

A few recent works have tackled the dynamic variants of the MaxRS problem [**7, 57**]. These works consider objects that may appear or disappear (i.e., insert/delete); however, the locations of the objects do not change over time. To the best of our knowledge, the Co-MaxRS problem has not been addressed in the literature so far. To solve the Co-MaxRS problem, we first identify the critical times at which the Co-MaxRS solution may change and a new solution appears. We show that there is only a certain *finite* number of such events possible, which enables us to propose a base algorithmic solution to the problem. Given the relatively expensive computational time (and worst-case time complexity) for the base algorithm, we devise several effective pruning strategies to minimize redundant computations and improve the overall processing time. We present a compact data structure to facilitate the pruning schemes, and use the concept of *Kinetic Data Structures* (KDS) to manage the critical events. Experiments over both real and synthetic datasets show an order of magnitude improvement in the processing time compared to the base algorithm and other possible approaches.

In many real-world applications, trajectories data sets need to reside on a secondary storage or even on cloud [**21**]. Even when stored in-memory, a sequential access to the whole trajectories is inefficient. To deal with these cases, we present a hierarchical grid-based index structure which can be built on-the-fly and maintained via KDS. The proposed grid-based indexing scheme can be utilized both as an in-memory and external memory structure. Moreover, we investigate the trade-offs between processing time vs. approximate answer to Co-MaxRS (similar to [**17**]). There are many scenarios where a fast response to the query is necessary, e.g., real-time data analysis via GUI (see [**37**]). For example, in case of **Q1** presented above, an approximate solution that is closer to the current MaxRS solution can be preferred more than an exact solution due to the logistics (time, battery, etc. needed) for the drone to move between locations. A fast approximate solution with a theoretical bound on the errors is desirable in this case. We present a novel approximation algorithm for the Co-MaxRS problem using the grid-based partitioning system. We prove that the approximation ratio for the algorithm is a small constant, i.e., 4, while the computational cost is an order of magnitude faster than the exact solution. Experimental analysis on various datasets demonstrate the benefits of our proposed approximation technique and index structure.

The main contribution of our work can be summarized as follows:

• We formally define the Co-MaxRS problem and identify criteria (i.e., *critical times*) under which a particular MaxRS solution may no longer be valid, or a new MaxRS solution emerges. These, in turn, enable algorithmic solution to Co-MaxRS using procedures executing at discrete time instants.

• Given the worst-case complexity of the problem (consequently, the algorithmic solution), we propose efficient pruning strategies to reduce the cost of recomputing the Co-MaxRS solutions at certain critical times. We present an in-memory data structure and identify properties that enable two such strategies: (1) eliminating the recomputation altogether at corresponding critical time; (2) reducing the number of objects that need to be considered when recomputing the Co-MaxRS solution at given critical times.

• We present a hierarchical grid-based indexing structure, specific to the Co-MaxRS problem, to improve objects retrieval time. The proposed index can be implemented both as an in-memory or external memory structure.

• Given that in some cases, an approximate solution to the Co-MaxRS problem is suitable – we devise a grid-based approximation algorithm. We show that the processing time of the offered solution is faster than the exact algorithm, while it has a constant approximation ratio of 4.

• We evaluate our proposed approaches using both real and synthetic datasets, and demonstrate that the pruning strategies yield much better performance than the worst-case theoretical bounds of the Co-MaxRS algorithm – e.g., we can eliminate 80-90% of the critical time events and prune ∼70% objects (on average) when recomputing Co-MaxRS. Additionally, we show that the approximation technique produces near-accurate results (e.g., ∼90% Avg. Accuracy) in very quick time (more than 100 times faster than the exact solution in some cases), while the indexing structure filters-out a good portion of the data before the pruning stage.

1.2. Condtional-MaxRS

Rapid advances in accuracy and miniaturization of location-aware devices, such as GPS-enabled smartphones, and increased use of social networks services (e.g., check-in updates) have enabled a generation of large volumes of spatial data [**50**]. In addition to the *(location, time)* values, that data is often associated with other contextual attributes. Numerous methods for effective processing of various queries of interest in such settings – e.g., range, (k) nearest neighbor, reverse nearest-neighbor, skyline, etc. – have been proposed in the literature [**39, 85, 92**].

One particular spatial query that has received recent attention is the, so called, *Maximizing Range-Sum (MaxRS)* [**17**], which can be specified as follows: given a set of weighted spatial-point objects O and a rectangle r with fixed dimensions (i.e., $a \times b$), MaxRS retrieves a location of r that maximizes the sum of the weights of the objects in its interior. Due to diverse applications of interest, variants of MaxRS [**7, 63, 79, 36, 24, 37**] have been recently addressed by the spatial database and sensor network communities.

1.2.1. Motivation and Challenges

What motivates this work is the observation that in many practical scenarios, the members of the given set O of objects can be of different types, e.g., if O is a set of restaurants, then a given $o_i \in O$ can belong to a different class from among fast-food, Asian, French, etc. Similarly, a vehicle can be a car, a truck, a motorcycle, and so on. In the settings where data can be classified in different (sub)categories, there might be class-based participation constraints when querying for the optimum region – i.e., a desired/minimum number of objects from particular classes inside r. However, due to updates in spatial databases – i.e., objects appearing and disappearing at different times – one needs to accommodate such dynamics too. Following two examples illustrate the problem:

Example 1: Consider a campaign scenario where a mobile headquarters has limited amount of staff and needs to be positioned for a period of time in a particular area. The US Census Bureau has multiple surveys on geographic distributions of income categories[1] and, for effective outreach purposes, the campaign managers would like to ensure that within the limited reachability from the headquarters, the staff has covered a maximum amount of voters – with the constraint that a minimum amount of representative from different categories are included. This would correspond to the following query:

Q1: *"What should be the position of the headquarters at time t so that at least κ_i residents from each income Category$_i$ can be reached, while maximizing the number of voters reached, during that campaign date."*

Example 2: Consider the scenario of X's **Loon Project**[2], where there are different types of users – premium (class A), regular (class B), and free (class C), and users can disconnect or reconnect anytime. In this context, consider the following query:

Q2: *"What should be the position of an Internet-providing balloon at time t to ensure that there are at least Θ_i users from each Class$_i$ inside the balloon-coverage and the*

[1]cf. https://www.census.gov/topics/income-poverty/income.html

[2]The Loon project (formerly Google X [**4**]) aims at providing internet access to remote/rural areas via a collection of high-altitude balloons providing wireless networks with up to 4G LTE speeds.

number of users in its coverage is maximized?".

It is not hard to adapt **Q1** and **Q2** to many other applications settings: – environmental tracking (e.g., optimizing a range-bounded continuous monitoring of different herds of animals with both highest density and diversity inside the region); – traffic monitoring (e.g., detecting ranges with densest trucks); – video-games (e.g., determining a position of maximal coverage in dynamic scenarios involving change of locations of players and different constraints).

We call such queries *Conditional Maximizing Range-Sum* (C-MaxRS) queries, a variant of the traditional MaxRS problem. For dynamic settings, where the objects can be inserted and/or deleted, we have *Conditional Maximizing Range-Sum with Data Updates* (C-MaxRS-DU) query.

An illustration for C-MaxRS query in a setting of 7 users grouped into 3 classes (i.e., A, B, and C), and with a query rectangle size $a \times b$ (i.e., height a and width b) is shown in Figure 1.2. Assume that the participation constraint is that *the positioning of* r *must be such that at least 1 user is included from each of the classes A, B, and C, respectively.* There are two rectangles r_1 and r_2, with dimension $a \times b$, that are candidates for the solution. However, upon closer inspection it turns out that although r_2 contains most users (corresponding to the traditional MaxRS solution), it is r_1 that is the sought-for solution for the C-MaxRS problem. Namely, r_2 does not satisfy the participation constraints (see Figure 1.2(i)).

Now, suppose that at time t_2, user o_6 disconnects and a new user o_8 joins the system. Then the C-MaxRS solution will need to be changed to r_2 from r_1 (see Figure 1.2(ii)).

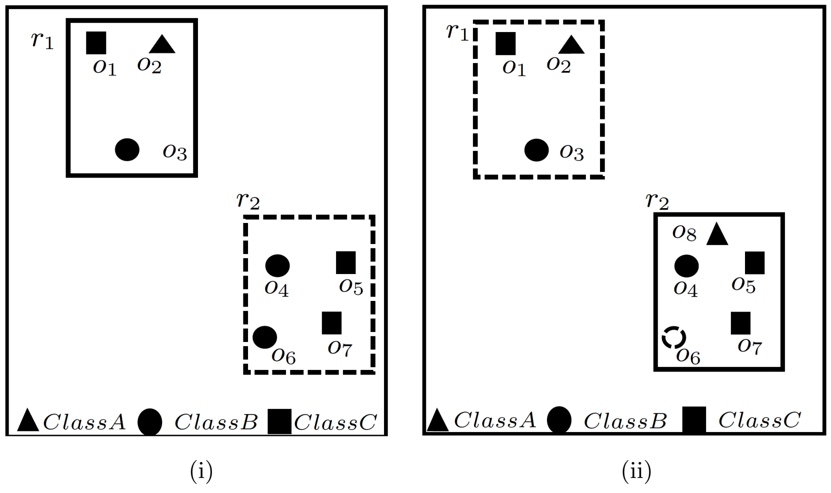


Figure 1.2. An example of C-MaxRS problem in spatial data streams at time (i) t_1 (ii) t_2.

1.2.2. Contributions

Our key idea for efficient C-MaxRS processing is to partition the space and apply effective pruning rules for each partition to quickly update the result(s). The basic processing scheme follows the technique of spatial subdivision from [**24**], dividing the space into a certain number of slices, whose local maximum points construct the candidate solution point set. In each slice, the subspace was divided into slabs which helps in reducing the solution space. To handle dynamic data stream scenarios, i.e., appearances and disappearances of objects, we propose two algorithms, *C-MaxRS*$^{+}$ and *C-MaxRS*$^{-}$ respectively, which works as a backbone for solving the constrained maximum range sum queries in the dynamic insertions/deletions settings (C-MaxRS-DU). Our novelty is in incorporating heuristics to reduce redundant calculations for the newly appeared or disappeared points, relying on two trees: a quadtree and a balanced binary search tree. Experiments over a wide range of parameters show that our approach outperforms the baseline algorithm by a factor of three to four, for both Gaussian and Uniform distribution of datasets.

The above idea for the C-MaxRS-DU algorithm takes an event-based approach, in the sense that C-MaxRS is evaluated (maintained) every time an event occurs, i.e., new point appears (e^+) or an old point disappears (e^-). This approach works efficiently when events are distributed fairly uniformly in the temporal domain and occur at different time instants that are enough apart for reevaluation to complete. However, the recent technological advancements and the availability of hand-held devices have enabled a large increase (or decrease) of the number of active/mobile users in multitude of location-aware applications in relatively short time-spans. In the context of Examples 1 and 2, this would correspond to the following scenarios:

Example 1': If the area involves businesses, then one would want to exploit the fact that many individuals may: (a) come (or leave) their place of work in the morning (or evening); (b) enter (or leave) restaurants during lunch-time; etc.

Example 2': In the settings of X's Loon Project, there can be multiple users disconnecting from the service simultaneously (within a short time span), or new users may request connections.

There are many other scenarios from different domains – e.g., Facebook has on average 2 billion daily active users – approximately 23,000 users per second. These Facebook users can be divided into many groups (classes), and C-MaxRS can be used to retrieve the most interesting regions (with respect to particular requirements) among the active daily users. In this scenario, a large number of users can become online (e^+), or go offline (e^-) at almost-same time instant. Similarly, flocks of different kinds of animals may be approaching the water/food source; the containment of the diseases across the population and regions may vary; etc.

To address the efficiency of processing in such settings, we propose a novel technique, namely *C-MaxRS-Bursty*. The key idea of our approach is as follows: instead of processing every single update, we assume that the update streams are gathered for a period of time. Then, we create a modified slice-based index for the entire batch of the new events, and then snap the new data over the existing slice structure in a single pass. Finally, we perform the pruning conditions for each slice only once in an aggregated manner. Experimental results show that C-MaxRS-Bursty outperforms our one-at-a-time approach, C-MaxRS-DU, by a speed-up factor of 5-10.

The main contributions of this work can be summarized as follows:

• We formally define the C-MaxRS and C-MaxRS-DU problems (for both weighted and non-weighted versions) and provide a baseline solution using spatial subdivision (slices).

• We extend the solution to deal with spatial data streams (appearing and disappearing objects) for which we utilize effective pruning schemes for both appearing and disappearing events, capitalizing on a self-balancing binary search tree (e.g., AVL-tree) and a quad-tree.

• We propose an efficient methodology to handle bulk updates of data (i.e., updates with large data-volumes) along with the appropriate extensions of the data structures to cater to such settings.

• We demonstrate the benefits of our proposed method via experiments over a large dataset. Experiments over a wide range of parameters show that our approaches outperform the baseline algorithms by a factor of three to four. Moreover, experiments with bulk updates demonstrate the effectiveness and scalability of C-MaxRS-Bursty over other techniques (e.g., C-MaxRS-DU).

1.3. MAxRS^3

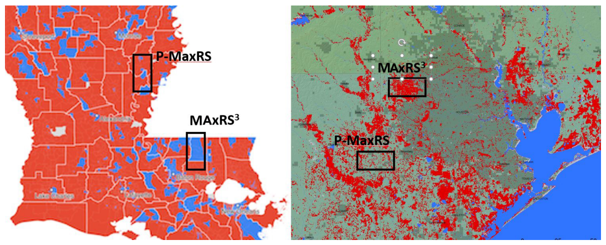


Figure 1.3. MAxRS^3 – Louisiana votes and Texas floods.

The Maximizing Range Sum query (MaxRS) takes a collection of weighted spatial point-objects O and a rectangle R with fixed dimensions as inputs, and generates a location(s) for placing the centroid of R that maximizes the sum of the (weights of the objects) in R's interior. Initially, the MaxRS problem was identified and solved by the researchers in computational geometry (CG) community [**58**]. Some years later, motivated by its importance in location-aware queries, such as: *what is the best location for a new franchise store with a limited delivery range*, or *what is the hotel location so that a tourist with spatially constrained mobility can see most attractions* – researchers have tackled various new aspects. Efficient solution for MaxRS in large (secondary storage) spatial databases has been presented in [**17**]; more recently, a continuous variant of MaxRS for mobile objects and query-rectangle has been addressed in [**32**], and dynamic settings where objects may be inserted/deleted along with changing their weights have also been considered (cf. [**7, 56**]).

We note that [**58**] considered a variation of the MaxRS problem where the input collection consists of polygons instead of points. For brevity, we call that variant a P-MaxRS (Polygons MaxRS), and a solution was presented so that the (weighted) sum of the polygons inside R is maximized.

1.3.1. Motivation and Contribution

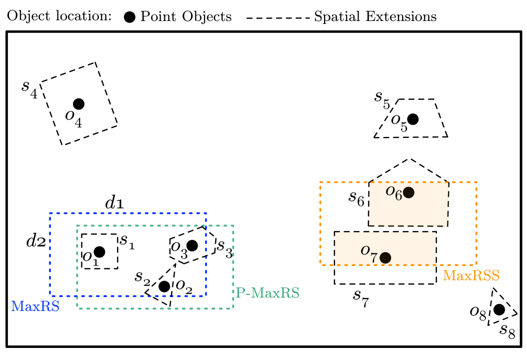


Figure 1.4. MaxRS vs. P-MaxRS vs. MAxRS^3.

What motivates our work is the observation that in many practical settings, in addition to the datasets consisting of polygons – it is more important to find a placement for the centroid of query-rectangle R in a manner that will ensure *maximal area coverage*. As specific examples, consider the following scenarios:

S1: A campaign manager with a limited reachability for his staff would like to know where to place the mobile headquarters to improve the votes in a given region.

S2: Emergency crews are interested in location for placing the sump-pumps with limited reachability of multiple hose, so that the drainage impact is maximized.

Both scenarios are illustrated in Fig. 1.3 (left portion is illustrating **S1** and the right portion illustrates **S2**). In each case, we show two positions of R: (1) covering maximal number of regions (i.e., P-MaxRS); and (2) covering maximal area. Clearly, the placement(s) based on the solution to P-MaxRS are not the desired output for **S1** and **S2**. To address such problems, in this paper we propose the MAxRS^3 (Maximal Area-Range Sum for Spatial Shapes) problem. More formally, the fundamental differences between MaxRS, Polygon Containment (a.k.a. P-MaxRS) and MAxRS^3 problem are illustrated in Figure 1.4. Assuming that 8 point objects are given ($o_1, o_2, \ldots, o_8$ in Figure 1.4) and the weights of all the objects are uniform, the placement of the rectangle R indicated in dotted blue line is the MaxRS solution (i.e., count=3). When objects have a spatial extent represented by a polygon, e.g., triangle, rectangle, pentagon, etc. (cf. $s_1, s_2, \ldots, s_8$ centered at the point objects $o_1, o_2, \ldots, o_8$ in Figure 1.4), the solution to the P-MaxRS problem [**58**]) is given by the dotted green line, where the placement of R completely encapsulates polygons s_1, s_2, and s_3 (i.e., count=3). The MAxRS[3] problem – addressing the more practical goal of maximizing the area of the coverage of R, will return the placement represented by the dotted orange line, overlapping s_6 and s_7.

1.4. Preliminaries

We now review the approaches for solving static MaxRS problem and introduce the concept of kinetic data structures that we subsequently use for solving Co-MaxRS.

1.4.1. MaxRS for Static Objects

Let $C(p, R)$ denote the region covered by an isothetic rectangle R, placed at a particular point p. Formally:

Definition 1. (MaxRS) *Given a set O of n spatial points $O = \{o_1, o_2, \ldots, o_n\}$, where each o_i associated with[3] a weight w_i , the answer to MaxRS query ($\mathbb{A}_{MaxRS}(O, R)$) retrieves a position p for placing the center of R, such that $\sum_{\{o_i \in (\, O \cap C(p,R))\}} w_i$ is maximal.*

$\sum_{\{o_i \in (\, O \cap C(p,R))\}} w_i$ is called the *score* of R located at p. If $\forall o_i \in O : w_i = 1$, we have the *count* variant, instances of which at different times are shown in Figure 1.1. Note

[3]One may also assume that the points in O are bounded within a rectangular area $\mathbb{F}$.

that there may be multiple solutions to the MaxRS problem, and in the case of ties – one can be chosen randomly, unless other ranking/preference criteria exist.

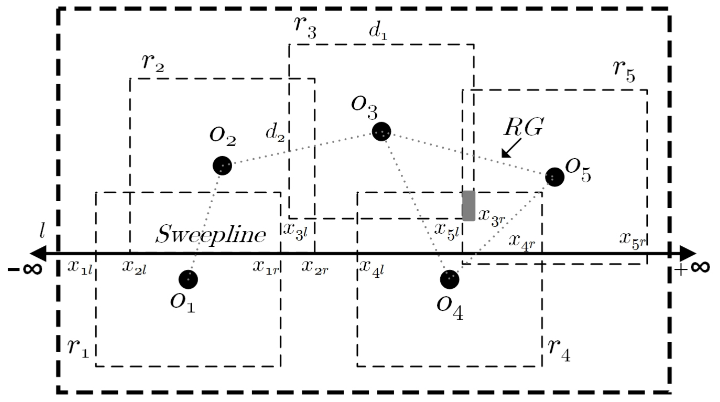


Figure 1.5. MaxRS $\rightarrow$ rectangle intersection.

Consider the example shown in Figure 1.5 – the count variant of MaxRS, with a rectangle R of size $d_1 \times d_2$ and five objects (black-filled circles). An in-memory solution to MaxRS (cf. [**58**]) transforms it into a "dual" *rectangle intersection problem* by replacing each object in $o_i \in O$ by a $d_1 \times d_2$ rectangle r_i, centered at o_i. R covers o_i if and only if its center is placed within r_i. Thus, the rectangle covering the maximum number of objects can be centered anywhere within the area containing a maximal number of intersecting dual rectangles (e.g., $r_3 \cap r_4 \cap r_5$ – gray-filled area in Figure 1.5).

Using this transformation, an in-memory algorithm to solve the MaxRS problem in $O(n \log n)$ time and $O(n)$ space was devised in [**58**]. Viewing the top and the bottom edges of each rectangle as horizontal intervals, an *interval tree* – i.e., a binary tree on the intervals – is constructed, and then a horizontal line is swept vertically, updating the tree at each event. The algorithm maintains the count for each interval currently residing in the tree, where the count of an interval represents the number of overlapping rectangles within that interval. When the sweep-line meets the bottom (top) edge of a rectangle, the corresponding interval is inserted to (deleted from) the interval tree and the count of each interval is updated accordingly. Considering the scenario in Figure 1.5 and using $[x_{il}, x_{ir}]$ to denote the left and right boundaries of r_i, when the horizontal sweep-line is at position l, there are 9 intervals: $[-\infty, x_{1l}]$, $[x_{1l}, x_{2l}]$, $[x_{2l}, x_{1r}]$, $[x_{1r}, x_{2r}]$, $[x_{2r}, x_{4l}]$, $[x_{4l}, x_{5l}]$, $[x_{5l}, x_{4r}]$, $[x_{4r}, x_{5r}]$, and $[x_{5r}, +\infty]$—with counts of 0, 1, 2, 1, 0, 1, 2, 1, and 0 respectively. An interval with the maximum count during the entire sweeping process is returned as the final solution and, since there can be at most $2n$ events (top or bottom horizontal edge of all r_i's) and each event takes $O(\log n)$ processing time, the whole algorithm takes $O(n \log n)$ time to complete.

We note that one may construct a graph RG (*rectangle graph*) where vertices correspond to points/objects in O (i.e., the centers of the dual rectangles) and an edge exists between two vertices o_i and o_j if and only if the corresponding dual rectangles overlap (i.e., $r_i \cap r_j \neq \emptyset$). As illustrated with dotted edges in Figure 1.5, an area of maximum overlap of dual rectangles corresponds to a maximum clique in RG.

1.4.2. Kinetic Data Structures

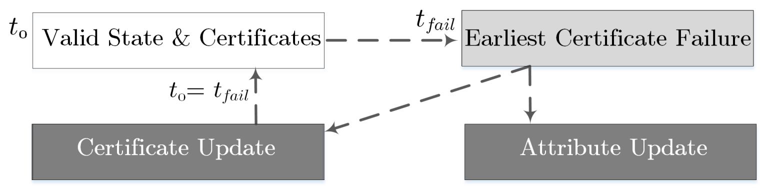


Figure 1.6. Kinetic Data Structures paradigm.

Kinetic data structures (KDS) [**11**] are used to track attributes of interest in a geometric system, where there is a set of values (e.g., location – x and y coordinates) that are changing as a function of time in a known manner. To process queries at a (virtual) current time t, an instance of the data structure at initial time t_0 is stored (i.e., values of the attributes of interest), which is augmented with a set of *certificates* proving its correctness at t_0. The next step is to compute the failure times of each certificates – called *events* – indicating that the data structure may no longer be an accurate representation of the state of the system. The events are stored in a priority queue sorted by their failure times. To advance to a time t $(= t_0 + \delta)$, we have to pop all the events having failure times $t_{fail} \leq t_0 + \delta$ from the queue in-order, and perform two operations at each event: (1) modify the data structure so that it is accurate at t_{fail} (attribute update), and (2) update the related certificates accordingly (see Figure 1.6). Throughout the dissertation paper, we utilize KDS in various ways, such as: (1) to maintain the Co-MaxRS answer-set over time and only perform certain tasks at the critical times (events) when a current MaxRS solution may change; and (2) to maintain the correct *count* for each grid of hierarchical grid-based partitioning.

1.5. Outline

In the rest of this paper, Chapter 2 positions the works with respect to the related literature, and offers brief description of some of our previous/related works. Chapter 3 describes our proposed algorithms to deal with the Co-MaxRS problem. Subsequently, Chapter 4 presents our solutions to the Conditional MaxRS problem in spatial data streams. In Chapter 5, we introduce the problem of MAxRS^3, and offer a novel and efficient algorithm to process it. Finally, Chapter 6 lays out current/ongoing works, future directions, proposed schedule, and overall contribution of the Ph.D. thesis.

CHAPTER 2

Related Literature

There are several bodies of research results that are closely related, and were used as foundation throughout this paper.

2.1. Related Works

Continuous MaxRS for Trajectories

The problem of MaxRS was first studied in the Computational Geometry community, with [**38**] proposing an in-memory algorithm to find a maximum clique of intersection graphs of rectangles in the plane. Subsequently, [**58**] used interval tree data structure to locate both (i) the maximum- and (ii) the minimum-point enclosing rectangle of a given dimension over a set of points. Although both works provide theoretically optimal bound, they are not suitable for large spatial databases, and a scalable external-memory algorithm – optimal in terms of the I/O complexity – was proposed in [**17**] (also addressing $(1 - \epsilon)$-*approximate MaxRS* and *All-MaxRS* problems). More recently, the problem of indexing spatial objects for efficient MaxRS processing was addressed in [**91**]. In this work, we used the method of [**58**] to recompute MaxRS only at certain KDS events, however, we proposed pruning strategies to reduce the number of such invocations. We note that an indexing scheme based on a static sub-division of the 2D plane (cf. [**17, 91**]) need not to be a good approach for spatio-temporal data because the densities in the spatial partitions will vary over time, and we plan to investigate the problem of efficient indexing techniques for Co-MaxRS as part of our future work.

In [**63**], an algorithm to process MaxRS queries when the locations of the objects are bounded by an underlying road network is presented. Complementary to this, in [**15**] the solution is proposed for the *rotating-MaxRS* problem, i.e., allowing non axis-parallel rectangles. Recently, [**7**] proposed methods to monitor MaxRS queries in spatial data streams – objects appear or disappear dynamically, but do not change their locations. A system implementing in-network solution to (a single, $k = 1$) MaxRS queries in WSN was presented in [**33**]. Although [**7**], [**15**], [**33**], and [**63**] deal with interesting variants of the traditional MaxRS problem, they do not consider the settings of mobile objects.

In this work, we relied on the KDS framework, introduced and practically evaluated in [**11**]. The KDS-like data structure was used to process critical events at which the current MaxRS solution may change. To estimate the quality of a KDS, [**11**] considered performance measures such as the time-complexity of processing KDS events and computing certificate failure times, the size of KDS, and bounds on the maximum number of events associated with an object. We used the same measures to evaluate the quality of our approach.

Circular (Co-)MaxRS: A special note is in order for the, so called, circular MaxRS [**14**] – which is, the region R is a disk instead of a rectangle. Arguably, this problem is $\Theta(n^2)$ and one of the main reasons is that the combinatorial complexity of the boundary of the intersection of a set of disks is not constant (unlike axes-parallel rectangles). This, in turn, would increase the $n \log n$ factor in our algorithms to n^2 – and the continuous variant of the circular MaxRS implies maintaining intersections of sheared cylinders instead of sheared boxes. We also note that this case (counting variant) bears resemblance to works that have tackled problems in trajectory clustering [**41**]. More specifically, [**25**] introduced the concept of flocks as a group of trajectories who are moving together within a given disk and for a given time, and [**40**] introduced the (less constrained) concept of trajectory convoys. These works, while similar in spirit to a continuous variant of the circular MaxRS – have not explicitly addressed the problem of detecting (and maintaining) the disk which contains the maximum number of moving objects, nor have considered weights of the objects. We re-iterate that the results in [**25**] show that some of the proposed algorithms have complexities similar in magnitude to the worst-case complexity of the Co-MaxRS. An approximate solution to the static variant of the circular MaxRS was presented in [**17**] (approximating the disk with the minimum bounding square) and our current Co-MaxRS solution can be readily applied towards the approximated variant.

MaxRS in Spatial Data Streams and Constrained Settings

Monitoring MaxRS for dynamic settings, where objects can be inserted and/or deleted was first addressed in [**7**]. To efficiently detect the new locations for placing the query rectangle, [**7**] exploited the aggregate graph $aG2$ in a grid index and devised a branch-and-bound algorithm [**59**] over that $aG2$ graph for efficient approximation. We note that our work is complementary to [**7**], in the sense that we addressed the settings of having different classes of objects and participation constraints based on them – whereas [**7**] solves the basic MaxRS problem. Moreover, [**7**] considered a sliding-window based model in the problem settings (i.e., if m new objects appear, then m old objects disappear in a time-window T), which is completely different to our event-based model. Additionally, we used contrasting approaches (and different data structures) in this work – dividing the 2D space into slices and slabs.

An interesting variant of MaxRS is addressed in [**24**] – the, so called, *Best Region Search* problem, which generalizes the MaxRS problem in the sense that the goal of placing the query rectangle is to maximize a broader class of aggregate functions[1]. Our work adapts the concepts from [**24**] (slices and pruning) – however, we tackle a different context: class-based participation constraints and dynamic/streaming data updates and, towards that, we also incorporated additional data structures (see Section 4.3). As a summary, our methodology (as well as the actual implementation) is based on the idea of event driven approach for monitoring appearing and disappearing cases of objects, and we included a self-balancing binary tree (i.e., AVL-tree) to reduce the processing time that is needed for computing the MaxRS as per the event queue needs.

Finally, a preliminary version of this work has been presented in [**55**]. However, we note that the techniques for processing continuous monitoring queries over data streams must be adaptive, as data streams are often bursty and input characteristics may vary over time. Many prior works have demonstrated the tendency of bursty streams in various applications, and proposed general solutions [**46, 9, 12**]. For example, [**9**] utilized ”load shedding” technique for aggregation queries over data streams, i.e., gracefully degrading performance when load is unmanageable; while [**12**] offered distributed stream processing systems to handle unpredictable changes in streaming rates. In this work, we address specifically the ”algorithmic” part of the problem, i.e., presenting an optimal processing technique for C-MaxRS during bursty inputs. Our proposed technique is

[1]More specifically, [**24**] was considering submodular monotonic functions as aggregates.

implementation-independent, and can be augmented by existing distributed and parallel schemes seamlessly (cf. Section 4.5).

We conclude this section with a note that few works [**20**], [**77**], and [**92**] studied the problem of maximizing reverse nearest neighbor, and [**82**] applied the optimal location query problem for road networks. However, we note that these problems are different from MaxRS as they do not consider maximum number of objects with interesting properties and instead consider the amount of influence only.

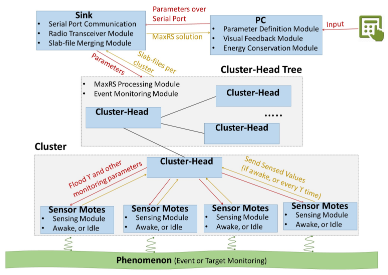
2.2. Our Previous Contributions

In the following, we describe a few of our previous contributions in more details that are related to the problem(s) solved in this thesis paper.

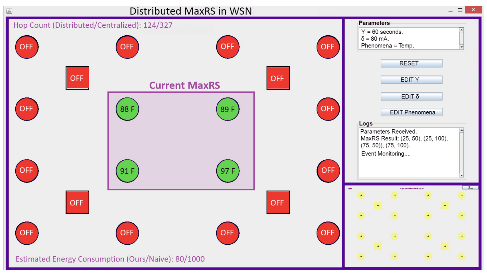
2.2.1. Distributed MaxRS in Wireless Sensor Networks

As we know, *Wireless Sensor Networks* (WSN) represent a paradigm with broad range of practical applications [**6**]. As sensor nodes are usually powered by batteries and often severely limited for recharging or replacing them, reducing the energy consumption is an ever-important topic in WSN, enabling an extension of overall network's operational lifetime [**8**]. Various works have also tackled routing issues, as well as epoch-based synchronization for query processing, aggregation, and in-network algorithms to minimize communication overheads (compared to centralized processing) [**49**]. Our goal in [**33**] was to provide efficient mechanisms for: (1) Processing MaxRS query in WSN; and (2) Reducing energy consumption while maintaining its answer under dynamically changing values of the monitored phenomena.

This work presented a distributed implementation for processing *Maximizing Range Sum* (MaxRS) query in Wireless Sensor Networks (WSN). As discussed in Chapter 1, MaxRS query is useful in many spatially-distributed event monitoring and target tracking applications. Given the location and current readings of the nodes, and a rectangle R, MaxRS finds a location of R that maximizes the sum of the readings of all the nodes covered by R. Our system performs MaxRS query in a user-specified time-interval γ and using the result obtained, attempts to maintain a certain degree of energy conservation in the WSN, based on a user-defined threshold δ. Since centralized processing of the raw readings and subsequently determining the MaxRS may incur significant communication overheads, we developed a distributed algorithm to compute MaxRS. We implemented our system in a heterogeneous WSN consisting of TelosB and SunSPOT motes, and illustrate the end-user tools: GUI for specifying required parameters, and real-time visualization of MaxRS solutions and estimated network energy consumption (cf. Figure 2.1b).



(a)



Hop Count (Distributed/Centralized): 124/327

Estimated Energy Consumption (Ours/Naive): 80/1000

Parameters

Υ = 60 seconds.
δ = 80 mA.
Phenomena = Temp.

Logs

Parameters Received.
MaxRS Result: (25, 50), (25, 100), (75, 50)), (75, 100).
Event Monitoring....

(b)

Figure 2.1. (a) System Design (b) Graphical User Interface.

The distributed solution for the MaxRS problem in spatial databases in [**17**] is the foundation of our in-network algorithm. Instead of dividing the space into m vertical slabs, we divide the space into a hierarchy of $m \times n$ grid-shaped clusters, each of which has a selected cluster-head. The system architecture shown in Figure 2.1a consists of the following main components:

(1) At the highest declarative level, we have the GUI that serves two main purposes: (a) Enables the users to select desired values for the required parameters; and (b) Provides a display for visualizing the boundaries of the current MaxRS solution, showing the nodes that are currently awake; the status of the monitored event (e.g., mote readings); and energy consumption in the network.

(2) The sink, connected through the serial port of the laptop: (a) Disseminates the parameters to the cluster-heads; and (b) Collects the data from the cluster-heads and performs the slab-file merging procedure described in [**17**].

(3) The cluster-heads form a tree-based hierarchy towards the sink. Each cluster-head calculates the slab-file information for the local cluster and coordinates the information with one of the neighboring cluster-heads to perform in-network aggregation.

(4) At the lowest level, the individual motes in each cluster conduct two simple tasks: (a) They receive related parameters from the local cluster-head; and (b) When awake, they report their readings to the local cluster-head.

2.2.2. k-MaxRS in Wireless Sensor Networks

Wireless Sensor Networks (WSN) consist of hundreds, or even thousands of nodes capable of sensing particular set of phenomena, performing basic computations and, most importantly, communicating with each other [**6**]. They are the empowering technology for a wide range of applications including environmental monitoring, smart buildings

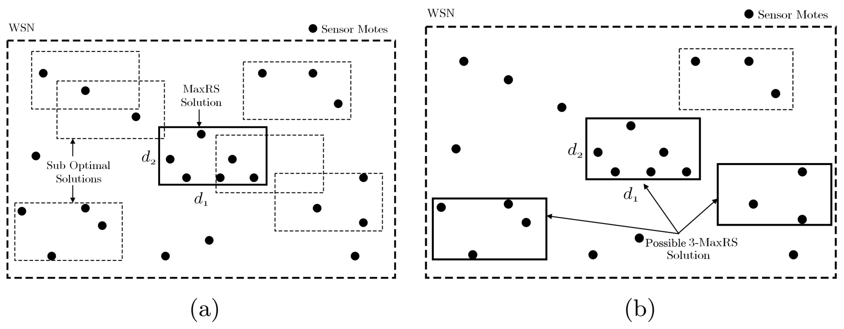


Figure 2.2. An example of (a) MaxRS and (b) k-MaxRS problem in WSN.

and cities, safety and hazard detection, agriculture, medicine, military, traffic monitoring, etc. Due to the types of sensors used and/or various deployment constraints (e.g., harsh and inaccessible environments), re-charging nodes' batteries is not always feasible. Consequently, reducing the energy consumption is an ever-important topic in WSN, facilitating an extension of overall network's operational lifetime [**8**]. While periodic sampling and transmission to a dedicated base-station may be applicable for certain applications, they may incur significant overhead in others – especially in event-based monitoring and tracking. To minimize communication overheads, various works have tackled coupling of routing schemes with aggregation and in-network query processing [**23, 48, 75, 49**].

In [**78**], we take a first step towards providing a distributed, energy-efficient solution in WSN settings, to the problem known as (k-)MaxRS – which can be described as follows. Given a collection of *weighted objects* O and a rectangle R with fixed dimensions (i.e., $d_1 \times d_2$), the *Maximizing Range Sum* (MaxRS) query retrieves the location at which (the centroid of) R should be placed, so that the sum of the weights of the objects in its interior is maximized. In the context of WSNs, we can think of the set of sensor nodes as the set of weighted objects, where the "weights" are application dependent, e.g., mote readings (event monitoring), information gain (tracking an object), uniform (counting), etc. We note that MaxRS is rather different from the traditional *range* query in the sense that when processing a range query, the region is typically fixed and one is interested in properties that hold in its interior. Contrary to this, MaxRS determines *where* should a rectangle with given dimensions be placed, so that some "interesting" properties in its interior are maximized (modulo all the other possible placements). An instance of the MaxRS in WSN is shown in Figure 2.2(a), assuming that the weights of all the sensor nodes are uniform – i.e., the "counting" variant.

Consider the following query:

Q1: *"Where should we place k surveillance devices (e.g., cameras, checkpoints, etc.) with a fixed-size coverage region in a forest such that their cumulative monitoring of forest-fire vulnerable regions (i.e., regions where temperature and light sensor readings are higher) is maximized?"*.

It is not hard to adapt **Q1** to other applications in which a simultaneous detection of top-k "popular" regions ($k \geq 2$) may be of interest. Such examples are: discerning k herds of tracked animals (e.g., gazelles) with largest density; aiding transportation system management by identifying k regions of the city with heaviest traffic; detecting congestions/hotspots in WSN by setting a node's current incoming/outgoing network traffic as its weight. One can also readily extrapolate to various collaborative scenarios – e.g., guiding drones towards regions where certain phenomenon has the largest weighted sum. To the best of our knowledge, the existing solutions [**34**] to MaxRS queries in WSN can only be applied to retrieve an optimal location for a *single* rectangle R, whereas **Q1** is an instance of the k-MaxRS variant – which we tackle in [**78**].

The respective k-MaxRS query finds the placement-locations for k rectangles such that the weighted sums of all the objects in the (union of the) interiors of each of R placed at those locations are optimal. An example-solution of the k-MaxRS query in WSN for uniformly weighted nodes is illustrated in Figure 2.2(b) (k=3). Although the MaxRS problem has been addressed by both computational geometry and spatial databases communities [**16, 17, 38, 58**], to the best of our knowledge, there has been no solution considering k optimal placements in WSN settings. As mentioned, in WSNs it is paramount to have energy-efficient query processing, for which in-network aggregation of partial results is often the approach of choice [**23, 48**]. Another challenge in this scenario is that the weights (i.e., sensor readings, information gain, etc.) of the sensor nodes may change with time, although their locations are fixed.

The main contribution of [**78**] can be summarized as follows:

• We provide an efficient in-network distributed algorithm to compute k-MaxRS via a hierarchy of clusters.

• We provide effective data-sharing schemes among the cluster-heads (also called principals).

• We provide experimental observations quantifying the benefits of the proposed approach.

2.2.3. Location and Weather Context in Place Recommendation Systems

In this work [**35**], we implemented a system for augmenting the functionality of Yelp-like recommendation sites by enabling users to search for places bounded by travel-time when using public transportation, and modifying recommendations based on updated weather conditions. Using public transport, although is cheaper and efficient, entails that only fixed places of boarding/exiting may be used which, in turn, implies walking to (from) a particular location from (to) a given station. Given the impact of the weather on the mood and activities, preferences for a certain type of services may need to be dynamically adjusted based on the current weather or the near-future forecast, modulo travel-routes to preferred locations. We developed a model to predict a user's preferred mode of transport (car, or public transit) from their old check-ins and incorporate the weather context into the recommendation process. We used event-based modeling to control the extent of walking depending on user-defined tolerance information and live weather conditions, and implemented a web application (both desktop and mobile platforms), utilizing existing tools such as Google Maps Direction API [**1**] and OpenWeatherMap API [**2**] for retrieving real-time information.

An efficient solution for Co-MaxRS provides several benefits, in different aspects of [**35**] as following:

• Using solutions from Co-MaxRS, we can retrieve hotspots for moving objects, e.g., cars, users, etc. Based on that, clever pre-computation and cache schemes can be utilized to improve the performance of online route information queries to Google Maps Direction API.

• We leveraged upon clustering schemes to develop a model to estimate how far each user is willing to travel. We apply Bayesian Information Criterion (BIC) to acquire the optimal number of clusters (i.e., k), and then employed k-means [**43**] method to obtain the desired clustering. As MaxRS retrieves the densest clusters for a given set of objects, related effective solutions can be utilized in this setting.

CHAPTER 3

Processing Continuous MaxRS for Trajectories

In this chapter, we first present a formal definition for the Co-MaxRS problem, and then provide a basic solution of $O(n^3 \log n)$. We then proceed to offer clever pruning schemes and refinement strategies to improve processing time. Subsequently, we present two extensions to our proposed solution to the Co-MaxRS problem: (1) Firstly, we devise an efficient indexing technique (can be used both as an in-memory and/or external memory solution) based on hierarchical grid-based partitioning to improve the processing time; (2) Secondly, using the similar indexing structure, we offer an approximate solution to the Co-MaxRS problem with a bounded approximation ratio of 4. Then, the details of our experiments and obtained results are described. Finally, we briefly present the details of a demonstration system for solving Co-MaxRS problem.

3.1. Basic Co-MaxRS

Interval tree was used as in-memory data structure of the planesweep algorithm in both [**58**] and the subsequent work addressing scalability [**17**]. However, these techniques cannot be straightforwardly extended to maintain MaxRS solutions continuously – i.e., one cannot expect to have an uncountably-infinite amount of interval trees (at each instant of objects' motion). As it turns out, the answer to Co-MaxRS can change only at discrete time-instants, which we address in the sequel.

Throughout this section, without loss of generality, we assume that each object moves along a single straight line-segment and all the objects start and finish their motion in the same time instant. We will lift this assumption and discuss its impact in Section 3.2.3.

Continuous MaxRS (Co-MaxRS) is defined as follows:

Definition 2. (Co-MaxRS) *Given a set O_m of n 2D moving points $O_m = \{o_1, o_2, \ldots, o_n\}$, where each is associated with a trajectory*[1] $o_i = [(x_{i1}, y_{i1}, t_{i1}), \ldots, (x_{i(k+1)}, y_{i(k+1)}, t_{i(k+1)})]$ *and a weight w_i; and a time-interval $T = [t_0, t_{max}]$, the answer to Co-MaxRS ($\mathbb{A}_{Co\text{-}MaxRS}(O_m, R, T)$) is a (time-ordered) sequence of pairs $[(l^1_{obj}, [t_0, t_1)), (l^2_{obj}, [t_1, t_2)), \ldots, (l^c_{obj}, [t_{c-1}, t_{max}))]$, where $(l^i_{obj}, [t_{i-1}, t_i))$ denotes the set of objects that determine the possible location(s) for R that is a MaxRS at any time instant $t_j \in [t_{i-1}, t_i)(\subseteq T)$.*

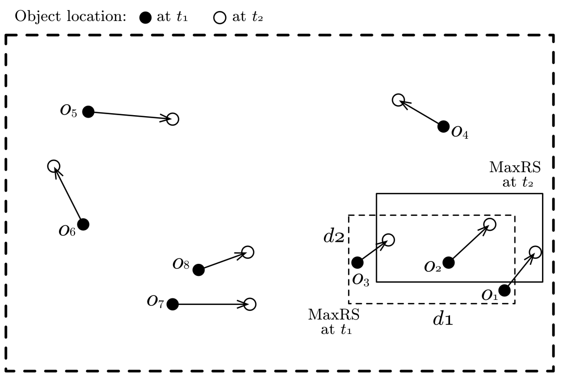


Figure 3.1. MaxRS location changes from t_1 to t_2, although the objects in the solution are the same.

Note that, instead of maintaining a centroid-location (equivalently, a region) as a Co-MaxRS solution, we maintain a list of objects that are located in the interior of the optimal

[1]Again, the trajectories may be bounded within a rectangular area $\mathbb{F}$.

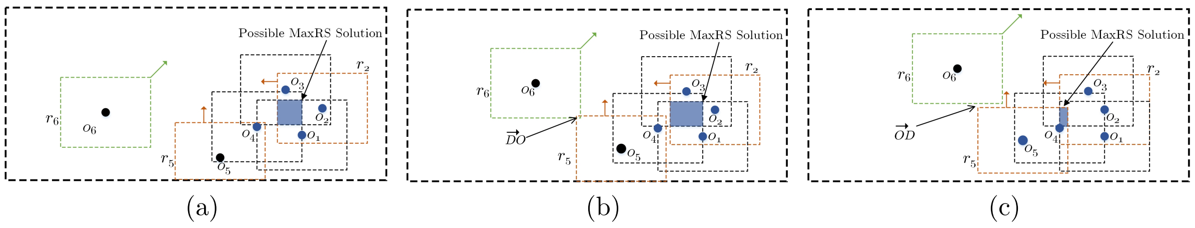


Figure 3.2. Co-MaxRS answer can only change when two rectangles' relationship changes from overlap to disjoint (or, vice-versa). Object locations at: (a) t_1 (b) t_2 (c) t_3.

rectangle placement. The rationale is two-fold: (1) Even for small object movements, the optimal location of the query rectangle can change while objects participating in the MaxRS solution stay the same; and (2) We can easily determine the trajectory (one of the uncountably-many) of the centroid of R throughout the time-interval during which the same set of objects constitutes the solution. An example is shown in Figure 3.1. At time t_1, objects o_1, o_2, and o_3 fall in the interior of the MaxRS solution. At t_2, although the same objects constitute the MaxRS solution, the optimal location itself has shifted due to the movement of the objects. Suppose there are s objects in the list l^j_{obj} at a particular time instant $t_s \in [t_{j-1}, t_j)$. Given l^j_{obj}, one can find the intersection of the s dual rectangles to retrieve the (boundaries of the possible) location for R at t_s in $O(s)$ time.

We can readily consider an alternative way of representing the Co-MaxRS solution – namely, as a trajectory of the (placement of the) centroid of R. Consider any time interval during which the same set of objects constitutes the solution – e.g., again $(l^j_{obj}, [t_{j-1}, t_j))$. Let $\{o^j_1, \ldots, o^j_s\}$ denote the actual objects from O_m defining l^j_{obj}. Their respective dual rectangles, $\{r^j_1, \ldots, r^j_s\}$ have a common intersecting region at $t = t_{j-1}$ – which, by assumption, is an axes-parallel rectangle. Every point in $\cap_{i=1}^{i=s} r^j_i$ can be a centroid of R

covering l^j_{obj} at t_{j-1}. Similarly for $t = t_j$ – once again we have an intersection of the s objects yielding an axes-parallel rectangle, except that both its size and location are changed with respect to the one at $t = t_{j-1}$. The key observations are:
(1) Each r^j_i dual rectangle, when moving along a straight line-segment (to follow the o^j_i) between t_{j-1} and t_j, "swipes" a volume corresponding to a sheared box/parallelopiped.
(2) At each $t \in [t_{j-1}, t_j)$ the intersection of the dual rectangles is non-empty (otherwise, it would contradict the fact that the objects in l^j_{obj} define the solution) and is a rectangle, thereby ensuring that the intersection of the parallelopipeds is continuously non-empty and, again, convex.

Thus, given the $\mathbb{A}_{MaxRS}(O, R)$ at $t = t_{j-1}$ and $t = t_j$, we can simply pick a point in the interior of each of the two (horizontal) rectangles in the (X, Y, Time) space, and the line-segment connecting them is one of the possible trajectories of the centroid of R as the solution/answer-set $\mathbb{A}_{\mathit{Co\text{-}MaxRS}}(O, R, T)$ (of course, for $T = [t_{j-1}, t_j)$).

We now describe how to identify when a recomputation of the MaxRS may (not) be needed due to the possibility of a change in the solution. Consider the example in Figure 3.2 with 6 objects: $\{o_1, o_2, \ldots, o_6\}$. Let r_i denote the dual rectangle for an object o_i. For simplicity of visualization, assume that only o_2, o_5 and o_6 are moving: o_2 in west, o_5 in north direction (orange rectangles and arrows), and o_6 in the northwest direction. Figure 3.2a, shows the locations of objects at t_1 and the current MaxRS solution, $l_{obj} = \{o_1,\ o_2,\ o_3,\ o_4\}$ (blue colored objects in Figure 3.2a). In this setting, r_2 and r_5 do not overlap. Figure 3.2b shows the objects' locations and their corresponding rectangles at t_2 ($> t_1$). Due to the movement of o_2 and o_5, the maximum overlapped area changed at t_2 (blue-shaded region). But, as r_2 and r_5 still do not overlap, the objects comprising the MaxRS solution are still the same as t_1. Finally, Figure 3.2c represents the objects' locations at a later time t_3, where r_2 and r_5 are overlapping. This causes a change in the list of objects making up the MaxRS solution, and o_5 is added to the current solution. We note that the solution changed only when two disjoint rectangles began to overlap. If we consider the example in reverse temporal order, i.e., assuming $t_3 < t_2 < t_1$, then the MaxRS solution changed when two overlapping rectangles became disjoint.

Observation: The solution of Co-MaxRS changes only when two rectangles change their topological relationship from *disjoint* to *overlapping* ($\vec{D}O$), or from *overlapping* to *disjoint* ($\vec{O}D$). We consider the objects along the boundary of the query rectangle R as being in its interior, i.e., rectangles having partially overlapping sides and/or overlapping vertices are considered to be *overlapping.* In the rest of this section, if we need to indicate an occurrence of $\vec{D}O$ or $\vec{O}D$ at a specific time instant t and pertaining to two specific objects o_i and o_j, we will extend the signature of the notation by adding time as a parameter and index the objects in the subscript (e.g., $\vec{D}O_{i,j}(t)$ or $\vec{O}D_{i,j}(t)$).

Thus, as the objects (resp. dual rectangles) move, there are two kinds of changes:

(1) *Continuous Deformation:* As the locations of the rectangles change, the overlapping rectangular regions may change, but the set of objects determining any overlapping rectangular region remains the same.

(2) *Topological Change:* Due to the movement of the rectangles, a $\vec{D}O$ or $\vec{O}D$ transition occurs for a pair of rectangles.

We note that, while the change of the topological relationship is necessary for a change in the answer set in the continuous variant of $\mathbb{A}_{MaxRS}(O_m, R)$ – it need not be sufficient. As shown in Figure 3.2, the relationship between r_5 and r_6 transitioned from disjoint, to overlap, and to disjoint again. However, none of those changes affected the $\mathbb{A}_{\textit{Co-MaxRS}}(O_m, R, T)$ between t_1 and t_3.

In Section 3.2 we will use this observation when investigating the options of pruning certain events corresponding to changes in topological relationships. At the time being, we summarize the steps for a brute-force algorithm for calculating the answer to Co-MaxRS:

Algorithm 1 Basic Co-MaxRS

```
        Input: $(O_m, R, T = [t_0, t_{max}])$
 1: Calculate all the time instants for all the pairwise topological changes for the objects
    in $O_m$
 2: Sort the times of topological changes
 3: For each such time $t_i^{tc}$, execute $\mathbb{A}_{MaxRS}(O, R)$
 4: **if** Objects defining the answer set are the same **then**
 5:     Extend the time-interval of the validity of the most recent entry in $\mathbb{A}_{\textit{Co-MaxRS}}$
        $(O_m, R, T = [t_0, t_{max}])$
 6: **else**
 7:     Close the time-interval of validity of the prior most-recent entry
 8:     Add a new element into $\mathbb{A}_{\textit{Co-MaxRS}}$ $(O_m, R, T = [t_0, t_{max}])$ consisting of the objects
        defining the $\mathbb{A}_{MaxRS}(O, R)$ at $t_i^{tc}$, with the interval $[t_i^{tc}, t_{i+1}^{tc})$
 9: **end if**
10: **return** $\mathbb{A}_{\textit{Co-MaxRS}}(O_m, R, T)$
```

Clearly, the complexity of Algorithm 1 is $O(n^3 \log n)$ – which can be broken into: – $O(n^2)$ for determining the (pairwise) times of topological changes; – $O(n^2 \log n^2)$ for sorting those times; – executing $O(n^2)$ times the instantaneous $\mathbb{A}_{MaxRS}(O, R)$ (at $O(n \log n)$). We note that $O(n^3 \log n)$ is actually a tight worst-case upper-bound, since the solutions in $\mathbb{A}_{MaxRS}(O, R)$ can be "jumping" from one R-region into another that is located elsewhere in the area of interest between any two successive intervals – which are $O(n^2)$.

3.2. Pruning in Co-MaxRS

Given the complexity of the naïve solution – which, again, captures the worst-case possible behavior of moving objects – we now focus on strategies that could reduce certain computational overheads, based on (possible) "localities". We discuss two such strategies aiming to: (1) Reduce the number of recomputations of MaxRS; and (2) Reduce the total number of objects considered when recomputing the MaxRS solution[2], and then present the algorithms that exploit those strategies.

Before proceeding with the details of the pruning, we describe the data structures used.

Figure 3.3 depicts the data structures used to maintain the Co-MaxRS answer-set based on the KDS framework. Strictly speaking, it consists of:

Object List (OL): A list for storing each object $o_i \in O$, with its current trajectory Tr_{o_i} (i.e., snapshots of location at t_0 and t_{max}), weight w_i, sum of weights of its neighbors in the rectangle graph $WN(o_i)$, and whether or not the object is part of the current MaxRS solution. Note that, o_j is neighbor of o_i if r_i and r_j overlap.

Kinetic Data Structure (KDS): Figure 3.3 illustrates the underlying KDS (event queue), and its relation with the OL. Each event $E^{t_k}_{i,j}$ is associated with a time t_k, where $t_0 < t_k < t_{max}$. KDS maintains an event queue, where the events are sorted according to the time-value. Each event entry $E^{t_k}_{i,j}$ has pointers to its related objects – two object *identifiers*, and the type of the event – ($\vec{D}O$ or $\vec{O}D$).

Adjacency Matrix (AdjMatrix): Represents the time-dependent rectangle graph RG, with its rows and columns corresponding to the vertices of RG (i.e., the objects from

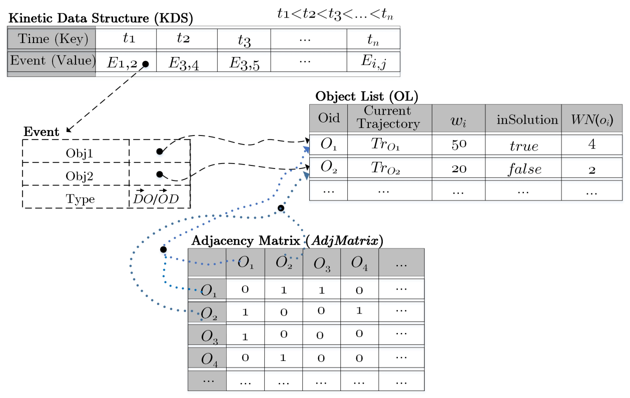
[2]Due to a lack of space, we do not present the proofs of the Lemmas in this work, however, they are available at [**51**].

O_m). For each pair of objects o_i and o_j, and a particular (critical) time instant, the $AdjMatrix[i][j]$ and $AdjMatrix[j][i]$ – set to 1 or 0 – indicate whether the two objects are directly connected with an edge in RG (i.e., their dual rectangles overlap).

3.2.1. Pruning KDS Events

Recall that the solution to MaxRS problem is equivalent to retrieving the maximum clique in the rectangle graph RG (cf. Section 1.4). For our first kind of pruning methodology, we leverage on the fact that a KDS event involving two objects o_i and o_j – which can be either $\vec{D}O_{ij}$ or $\vec{O}D_{ij}$ – is equivalent to adding or deleting an edge only between r_i and r_j, and no other objects/rectangles are involved. The properties that allow us to filter out $\vec{D}O$ and/or $\vec{O}D$ types of events without recomputing the MaxRS are discussed next.

$\vec{\boldsymbol{D}}\boldsymbol{O}$: Let $WN(o_i)(t)$ denote the current sum of the weights of the neighbors of an object o_i at time t, and let $score_{max}(t) = score((\mathbb{A}_{MaxRS}(O, R)), t)$ denote the score of the current MaxRS solution at t. During a $\vec{\boldsymbol{D}}\boldsymbol{O}$ event, the lower bound of a MaxRS solution is


Kinetic Data Structure (KDS)
$t_1 < t_2 < t_3 < \ldots < t_n$
Event
Object List (OL)
Adjacency Matrix ($AdjMatrix$)

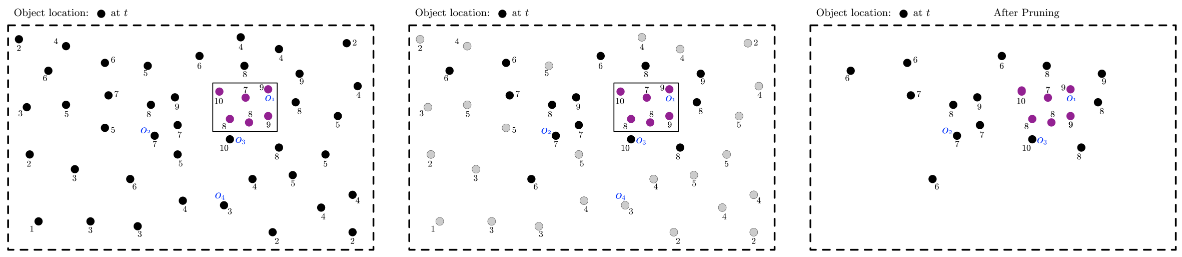
Time (Key)	t_1	t_2	t_3	...	t_n
Event (Value)	$E_{1,2}$	$E_{3,4}$	$E_{3,5}$	...	$E_{i,j}$

Obj1	
Obj2	
Type	$\vec{D}O/\vec{O}D$

Oid	Current Trajectory	w_i	inSolution	$WN(o_i)$
O_1	Tr_{O_1}	50	*true*	4
O_2	Tr_{O_2}	20	*false*	2
...	...	...	...	...

	O_1	O_2	O_3	O_4	...
O_1	0	1	1	0	...
O_2	1	0	0	1	...
O_3	1	0	0	0	...
O_4	0	1	0	0	...
...	...	...	...	...	...

Figure 3.3. Data structures used.


Object location: ● at t
Object location: ● at t
Object location: ● at t
After Pruning
o_1
o_2
o_3
o_4

(a) (b) (c)

Figure 3.4. An example showing the objects pruning scheme: (a) Objects' locations and $WN(o_i)$ values at t (b) Grey objects can be pruned using Lemma 3 in a $\vec{D}O$ event (c) Remaining objects after pruning at a $\vec{D}O$ event.

$score_{max}(t)$, and upper bound of the score (i.e., maximum possible score) of an overlapping region including an object o_i is $(WN(o_i) + w_i)$.

Lemma 1. *Consider the event $\vec{D}O_{i,j}$ for two objects o_i and o_j, occurring at time $t_{i,j}$. Let $l_{obj}^{(t_{i,j}-\delta)}$ (for some small δ) denote the Co-MaxRS solution just before $t_{i,j}$. After updating $WN(o_i)$ and $WN(o_j)$ at $t_{i,j}$ (i.e., because of $\vec{D}O_{i,j}$), $l_{obj}^{(t_{i,j}-\delta)}$ remains a MaxRS if one of the following two inequalities holds:*

(1) $WN(o_i)(t_{i,j}) + w_i \leq score_{max}(t_{i,j} - \delta)$

(2) $WN(o_j)(t_{i,j}) + w_j \leq score_{max}(t_{i,j} - \delta)$

$\vec{O}D$: In this case the intuition is much simpler – the score/count of an instantaneous MaxRS solution can only decrease (or, remain same) during an $\vec{O}D$ event, and if it decreases (i.e., changes), both of the objects involved in the event must have been in l_{obj}. Thus, we have:

Lemma 2. *Consider the event $\vec{O}D_{ij}$ for two objects o_i and o_j occurring at time $t_{i,j}$. Let $l_{obj}^{(t_{i,j}-\delta)}$ (for some small δ) be the current MaxRS solution before $t_{i,j}$. If one of the following two conditions holds:*

(1) $o_i \notin l_{obj}^{(t_{i,j}-\delta)}$

(2) $o_j \notin l_{obj}^{(t_{i,j}-\delta)}$

then $l_{obj}^{(t_{i,j}-\delta)}$ *remains a MaxRS solution after* $\vec{O}D_{ij}$ *(i.e, after* $t_{i,j}$*).*

To utilize Lemma 1 and 2, we maintain for each $o_i \in O_m$ the value of $WN(o_i)$, and whether or not the object is part of the current MaxRS solution. In Figure 3.3, two variables *inSolution* and $WN(o_i)$ are used for this purpose, updated accordingly during the processing of $\vec{D}O$ and $\vec{O}D$ events.

3.2.2. Objects Pruning

After filtering out many of the recomputations (Lemma 1 and Lemma 2), it is desirable to reduce the number of objects required in the recomputation. Towards that, we the following observations: (1) $WN(o_i) + w_i$ is an upper bound on possible MaxRS scores containing an object o_i; (2) $score_{max}$, the current MaxRS score, is a lower bound on possible MaxRS scores after a $\vec{D}O$ event; and (3) $score_{max} - min\{w_i, w_j\}$ is a lower bound on possible MaxRS scores after a qualifying $\vec{O}D_{ij}$ event. Let $E_{i,j}$ denote any event involving two objects o_i and o_j (be it $\vec{D}O_{ij}$ or $\vec{O}D_{ij}$). We have:

Lemma 3. *After updating* $WN(o_i)$ *and* $WN(o_j)$ *at* E_{ij}*, an object* o_k *can be pruned before recomputing MaxRS if one of the following two conditions holds:*

(1) $E_{i,j}$ *is a* $\vec{D}O$ *event and* $WN(o_k) + w_k \leq score_{max}$

(2) $E_{i,j}$ *is an* $\vec{O}D$ *event and* $WN(o_k) + w_k \leq score_{max} - min\{w_i, w_j\}$

Example 1. *Figure 3.4a demonstrates an example scenario with 46 objects. For the sake of simplicity, we only consider the counting variant (i.e.,* $\forall o_i \in O : w_i = 1$*) in this* *example. The count of neighbors (i.e., $WN(o_i)$) for each object is shown as a label, and the current MaxRS solution is illustrated by a solid rectangle where $score_{max}$ (or, $count_{max}$) = 6. Members of l_{obj} are colored purple in Figure 3.4. Some of the objects are marked with an id (e.g., o_1, o_2, o_3, and o_4), so that they can be identified clearly in the text. In this scenario, to process any event, we will first update the appropriate $WN(o_i)$ and $inSolution$ values. Then, suppose a new $\vec{D}O$ event is processed for one of the objects for which $WN(o_i) \leq 5$, e.g., between o_3 and o_4. Then that event will be pruned and MaxRS answer-set will remain the same as the maximum possible count of a MaxRS including that object will be $(5 + 1)$=6. Similarly, any $\vec{O}D$ event involving an object other than the purple ones would be filtered out. Figure 3.4b illustrates the application of Lemma 3, based on which all the objects in grey can be pruned during a $\vec{D}O$ event before recomputing MaxRS. Thus, after applying Lemma 3, we can prune 26 objects in linear time, i.e., going through the set of objects once and verifying the respective conditions. After pruning, 20 objects will remain (cf. Figure 3.4b) – only 43% of the total objects.*

According to Lemma 1, a $\vec{D}O_{ij}$ event is not pruned when both $WN(o_i) + w_i > score_{max}$ and $WN(o_j) + w_j > score_{max}$ hold. Let us use $N(o_i)$ to denote the list of neighbors of any object o_i. Additionally, we employ $CN(o_i, o_j)$ to represent common neighbors of two objects o_i and o_j, i.e., $N(o_i) \cap N(o_j)$. In this setting, there are two possible cases:

Case 1: *Both* $o_i, o_j \notin l_{obj}$. The observation here is that if there exists a new MaxRS solution at a $\vec{D}O_{ij}$ event, then both o_i and o_j must be present in the new solution as only they are affected by the new $\vec{D}O$ event – all other objects (and their related attributes) remain the same. Additionally, for any MaxRS solution including both o_i and o_j, only the members of $CN(o_i, o_j)$ can be in l_{obj}.

Case 2: *Either $o_i \in l_{obj}$ or $o_j \in l_{obj}$.* Let us assume $o_i \in l_{obj}$. Then, if o_j overlaps with all objects $o_k \in l_{obj}$ (an $O(|l_{obj}|)$ check), then we can directly have a new MaxRS solution including o_j, i.e., $l_{obj} = l_{obj} \cup o_j$. If this check fails, we can follow the similar procedure as case 1. Note that, the case of both $o_i, o_j \in l_{obj}$ is not possible as it contradicts the concept of $\vec{D}O_{ij}$ event, i.e., o_i and o_j are mutually disjoint before $\vec{D}O_{ij}$. Based on the above observations, we have the following two lemmas:

Lemma 4. *For an event $\vec{D}O_{ij}$ involving two objects o_i and o_j, we can prune all the objects except o_i, o_j, and $CN(o_i, o_j)$ before recomputing MaxRS.*

Lemma 5. *For an event $\vec{D}O_{ij}$ involving two objects o_i and o_j where $o_i \in l_{obj}$, we can set $l_{obj} \cup o_j$ as the new MaxRS solution if o_j overlaps with all objects $o_k \in l_{obj}$.*

To take advantage of Lemma 4, we need to keep track of neighbors of all the objects in addition to $WN(o_i)$, which is the purpose of the adjacency matrix (*AdjMatrix* in Figure 3.3). We note that one could also maintain a list $N(o_i)$ for each object – however, although each approach would incur $O(n^2)$ space overhead in the worst case, the adjacency matrix has certain advantages:

- Updating of the matrix information can be done in $O(1)$ time. For example, at a $\vec{D}O_{i,j}$ event we can directly set *AdjMatrix*[*i*][*j*]=1 and *AdjMatrix*[*j*][*i*]=1. Similarly, *AdjMatrix*[*i*][*j*] and *AdjMatrix*[*j*][*i*] can be set to 0 at an $\vec{O}D_{i,j}$ event.
- We can compute $CN(o_i, o_j)$ for two objects o_i and o_j efficiently by doing a *bit-wise AND* operation over *AdjMatrix*[*i*] and *AdjMatrix*[*j*].

Example 2. *Suppose there is a new $\vec{D}O$ event between objects o_1 and o_3 in the example in Figure 3.4. The event will not be pruned because both $WN(o_1)$ and $WN(o_3)$ ¿ 5. As*

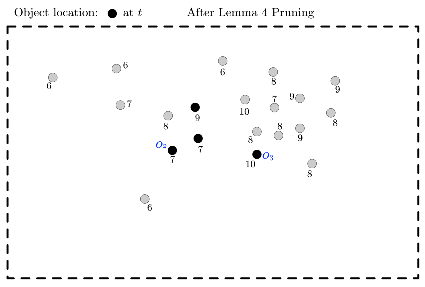


Figure 3.5. Application of Lemma 4 in $\vec{DO}$ events.

$o_1 \in l_{obj}$, we will first check if o_3 overlaps with all other members of l_{obj} (purple colored objects). As it does overlap with all the members of l_{obj}, we can directly output $l_{obj} \cup o_3$ as the new solution using Lemma 5. On the other hand, suppose the new $\vec{DO}$ occurs between o_2 and o_3. Using Lemma 4, we can prune all the objects except o_2, o_3, and $N(o_2) \cap N(o_3)$. This leaves us with only 4 remaining objects (cf. Figure 3.5) – 91.3% objects are pruned from the calculation. Obviously, score of the recomputed MaxRS will be less than the $score_{max}$ we already have (i.e., 6), and thus no change to the solution of Co-MaxRS will be made. We can see, Lemma 4 and Lemma 5 greatly optimizes processing of $\vec{DO}$ events.

3.2.3. KDS Properties and Algorithmic Details

Instead of a single line-segment, moving objects trajectories in practice are often polylines with vertices corresponding to actual location-samples. To cater to this, we introduce another kind of event, pertaining to an individual object – *line-change* event at a given time instant, denoted as $E_{lc}(o_i, t_{l_i})$. Suppose, for a given object o_i, we have $k+1$ time-samples during the period T as $t_{i1}, t_{i2}, \ldots, t_{i(k+1)}$, forming k line-segments. Note that the frequency of location updates may vary for different objects; even for a single object, the consecutive time-samples may have different time-gap. Initially, we insert the second time-samples for all the objects into the KDS as line-change events (cf. Figure 3.3). When processing $E_{lc}(o_i, t_{l_i})$ we need to compute: (a) Next $\vec{O}D$ events with the neighbors; and (b) Next $\vec{D}O$ events with other non-neighboring objects. We also need to insert a new line-change event at $t_{l_{(i+1)}}$ for o_i into the KDS. Thus, processing a line-change event takes $O(n)$ time. Note that a particular trajectory may start (appear) and/or finish its trip (disappear) at any time t, where $t_0 < t < t_{max}$ and we can use similar ideas to handle these special cases in $O(n)$ time.

KDS Properties: We proceed with briefly analyzing the properties of our proposed KDS-based structure (in the spirit of [**11**]), which shows that our adaption of KDS is responsive, efficient, local, and practically responsive.

(1) Number of certificates altered during an event (Responsiveness): Recall that we have two kinds of core events:

$\vec{D}O$ *Event:* At such an event we need to compute the time of the next $\vec{O}D$ event between the two objects and insert that to KDS if it falls within the given time-period T. Thus, only one new event (certificate) is added.

$\vec{O}D$ *Event:* For these events, we just need to process them, and no new event is inserted into KDS.

In both cases, the number is a small constant – conforming with the desideratum.

(2) The size of KDS (Compactness): In case of our adaptation of the KDS, we can have at most $O(n^2)$ $\vec{D}O$ and $\vec{O}D$ events at once. If we consider the additional line-change events for the polyline moving objects trajectories, there can be one such event for each object at any particular time, i.e., $O(n)$ such events. Thus, the size of KDS at a particular time is at most $O(n^2)$. However, as we will see in Section 3.5, in practice the size (total events) can be significantly smaller than this upper-bound – meeting the desideratum, i.e., $O(n^\epsilon)$ for some arbitrarily small $\epsilon > 0$.

(3) The ratio of internal and external events (Efficiency): In our KDS, the $\vec{DO}$ and $\vec{OD}$ events are external events (i.e., possibly causing changes to the Co-MaxRS answer-set), and the line-change events are internal. Thus, the ratio between total number of events and external events is $\frac{O(n^2)+O(n)}{O(n^2)}$, which is relatively small. This is a desired property of an *efficient* KDS [**11**].

(4) Number of certificates associated with an object (Locality): An object can have $n - 1$ $\vec{DO}$ and $\vec{OD}$ events with the other objects, and 1 line-change event at a particular time instant, i.e., the number of events associated with an object is $O(n)$, which is an acceptable bound.

Subsequently, our adaption of KDS is responsive, efficient, local and, in practice, compact too.

Algorithmic Details: In Algorithm 2, we present the detailed method for maintaining Co-MaxRS for a given time period $[t_0, t_{max}]$. As mentioned, for each object, in addition to *WN* and *inSolution* variables, we also keep track of the active neighbors in RG via *AdjMatrix*. After initialization (line 1 and 2), the KDS is populated with all the initial events that fall within the given time-period (line 3) – a step taking $O(n^2)$ time. Then, we retrieve the current solution, i.e., the list of objects, and create a new time-interval of its validity, starting at t^{new}_{start} in lines 4-6. We update the *inSolution* values of related objects whenever we compute a new MaxRS solution, and discard an old one (lines 7, 15, and 16).

Algorithm 2 Co-MaxRS (OL, R, t_0, t_{max})

```
 1: $KDS \leftarrow$ An empty priority queue of events
 2: $\mathbb{A}_{Co\text{-}MaxRS} \leftarrow$ An empty list of answers
 3: Compute next event $E_{next}$, $\forall o_i \in OL$ and push to $KDS$
 4: current $\leftarrow$ Snapshot of object locations at $t_0$
 5: $(loc_{opt}, score_{max}, l_{obj}) \leftarrow$R_Location_MaxRS (current)
 6: $t^{new}_{start} \leftarrow t_0$
 7: Update $inSolution$ variable for each $o_i$ in $l_{obj}$
 8: while $KDS$ not EMPTY do
 9:     $E_{i,j} \leftarrow KDS.Pop()$
10:     $(l'_{obj}, score_{max}) \leftarrow$ EventProcess $(E_{i,j}, KDS, l_{obj},$
        $score_{max})$
11:     if $l_{obj} \neq l'_{obj}$ then
12:         $t^{new}_{end} \leftarrow t_i$
13:         $\mathbb{A}_{Co\text{-}MaxRS}.Add(l_{obj}, [t^{new}_{start}, t^{new}_{end}))$
14:         $t^{new}_{start} \leftarrow t_i$
15:         Update $inSolution$ variable for each $o_i$ in $l_{obj}$
16:         Update $inSolution$ variable for each $o_i$ in $l'_{obj}$
17:         $l_{obj} \leftarrow l'_{obj}$
18:     end if
19: end while
20: $t^{new}_{end} \leftarrow t_{max}$
21: $\mathbb{A}_{Co\text{-}MaxRS}.Add(l_{obj}, [t^{new}_{start}, t^{new}_{end}))$
22: return $\mathbb{A}_{Co\text{-}MaxRS}$
```

Lines 8–19 process all the events in the KDS in order of their time-value, and maintain the Co-MaxRS answer-set throughout. The top event from the KDS is selected and processed using the function *EventProcess* (elaborated in Algorithm 3). After checking whether a new solution has been returned from *EventProcess*, the answer-set is adjusted in the sense of closing its interval of validity (t^{new}_{end}) which, along with the corresponding l_{obj} are appended to $\mathbb{A}_{Co\text{-}MaxRS}(O_m, R, T)$ (for brevity, the ".Add()" notation is used). A modified version of the MaxRS algorithm from [**58**] is used where, in addition to the score, the list l_{obj} is also returned – cf. *R_Location_MaxRS* in line 5. Note that, the condition check at line 11 in implementation actually takes constant time, which we detect via setting a boolean variable during MaxRS computation.

The processing of a given KDS event $E_{i,j}$ is shown in Algorithm 3. In line 1, the WN of the relevant objects and $AdjMatrix$ are updated. In lines 2–7, we compute new $\vec{O}D$ events and update the KDS. Lines 8–13 implement the ideas of Lemma 1 and Lemma 2, which takes $O(1)$ time. Lines 14–19 implement the idea of Lemma 5 to process a special kind of $\vec{D}O$ events. Line 20 introduces a new list OL', which will eventually retain only the unpruned objects. Lines 21-24 employ the idea of Lemma 4 for $\vec{D}O$ events. Lines 25–29 implement the ideas of objects pruning (Lemma 3), which takes $O(n)$ time. Finally, MaxRS is recomputed in lines 30–31 based on the current snapshot of the remaining moving objects in $O(n \log n)$ time (for brevity, we omitted handling line-change events in Algorithm 3). Lines 32–34 ensure that only valid computed values are returned, i.e., when $score'_{max} > score_{max}$ for $\vec{D}O$ events.

KDS Overhead Analysis: We ignored the cost of maintaining KDS in the above analysis, as it is subsumed by the time-complexity of Algorithm 3. The maintenance cost of KDS is related to the associated *event queue* – which is a priority queue of events. A priority queue can be implemented via heap, resulting in $O(\log n)$ insert (push) and delete minimum (pop) operations. In the worst case scenario, there can be $O(n^2)$ insertions and deletions in the KDS. The cost of $\{1, 2, 3, \ldots, n, \ldots, n^2\}$-th operation in the KDS would be $\{O(\log 1), O(\log 2), O(\log 3), \ldots, O(\log n), \ldots, O(\log n^2)\}$ – respectively. Thus, the total

Algorithm 3 EventProcess ($E_{i,j}$, KDS, l_{obj}, $score_{max}$)

```
1: Update $WN(o_i)$, $WN(o_j)$, and $AdjMatrix$ accordingly
2: if $E_{i,j}.Type = \vec{D}O$ then
3:     Compute $E_{next}$ for objects $o_i$ and $o_j$
4:     if $E_{next} \neq$ NULL and $E_{next}.t \in [t_0, t_{max}]$ then
5:         $KDS.Push(E_{next})$
6:     end if
7: end if
8: if $E_{i,j}.Type = \vec{D}O$ and ($WN(o_i) + w_i \leq score_{max}$ or $WN(o_j) + w_j \leq score_{max}$) then
9:     return ($l_{obj}$, $score_{max}$)
10: end if
11: if $E_{i,j}.Type = \vec{O}D$ and ($o_i.inSolution = false$ or $o_j.inSolution = false$) then
12:     return ($l_{obj}$, $score_{max}$)
13: end if
14: if $E_{i,j}.Type = \vec{D}O$ and Either $o_i/o_j \in l_{obj}$ then
15:     $o_k \leftarrow o_j/o_i$
16:     if $o_k$ and $l_{obj}$ are mutually overlapping then
17:         return ($l_{obj} \cup o_k$, $score_{max} + w_k$)
18:     end if
19: end if
20: $OL' \leftarrow OL$
21: if $E_{i,j}.Type = \vec{D}O$ then
22:     $CN(o_i, o_j) \leftarrow$ Compute-CN ($AdjMatrix$, $o_i$, $o_j$)
23:     $OL' \leftarrow CN(o_i, o_j) \cup \{o_i, o_j\}$
24: end if
25: for all $o_k$ in $OL'$ do
26:     if ($E_{i,j}.Type = \vec{D}O$ and $WN(o_k) + w_k \leq score_{max}$) or ($E_{i,j}.Type = \vec{O}D$ and
        $WN(o_k) + w_k \leq score_{max} - min(w_i, w_j)$) then
27:         Prune $o_k$
28:     end if
29: end for
30: current $\leftarrow$ Snapshot of objects in $OL'$ at $t_i$
31: ($loc'_{opt}, score'_{max}, l'_{obj}$) $\leftarrow$R_Location_MaxRS(current)
32: if ($E_{i,j}.Type = \vec{O}D$) or ($E_{i,j}.Type = \vec{D}O$ and $score'_{max} > score_{max}$) then
33:     return ($l'_{obj}$, $score'_{max}$)
34: end if
35: return ($l_{obj}$, $score_{max}$)
```

cost of maintaining the event queue would be:

$$O(\log 1 + \log 2 + \log 3 + \ldots + \log n^2)$$

$$\Rightarrow O(\log(1.2.3.\ldots.n^2))$$

$$\Rightarrow O(\log(n^2!))$$

We know that, $O(\log(n^2!))$ is asymptotically equivalent to $O(n^2 \log n^2)$ (using Stirling's Approximation of $n!$, [**65**]). Thus, the total cost of maintenance of Co-MaxRS queries for single-line trajectories is $O(n^3 \log n) + O(n^2 \log n^2)$ in the worst case.

Discussion: In the worst-case, Co-MaxRS for n trajectories with k segments throughout the query time-interval, has $O(kn^2)$ events. In KDS, $O(n^2)$ events are added at the beginning, then at each of the $O(kn)$ line change events, $O(n)$ new events may be created, resulting in $O(kn^2)$ events in total. Observe that between two consecutive event-times t_{s-1} and t_s, there is a Co-MaxRS path of constant complexity (i.e., the centroid of R moves along a straight line-segment). As mentioned in Section 3.1, this follows from the fact that the Co-MaxRS solution covering a particular list l^s_{obj} in the sequence $(\mathbb{A}_{\textit{Co-MaxRS}}(O_m, R, T))$ for the interval $[t_{s-1}, t_s]$, is the (maximum) intersection of sheared-boxes generated by the motion of the dual rectangles of the objects in l^s_{obj}. Thus, the worst-case combinatorial complexity of the path of the centroid of the Co-MaxRS solutions is $O(kn^2)$ – with a note that there may be discontinuities between consecutive locations of the centroids (i.e., the solution "jumps" from one location to another). The overall worst-case complexity when considering trajectories with multiple segments (i.e., polyline routes) is $O(kn^3 \log n)$.

We close this section with two notes:

(1) While the worst-case complexity of processing Co-MaxRS is high, such orders of magnitude are not uncommon for similar types of problems – i.e., detecting and maintaining flocks of trajectories [**25**]. However, as our experiments will demonstrate, the pruning strategies that we proposed can significantly reduce the running time.
(2) A typical query processing approach would involve *filtering* prior to applying pruning – for which an appropriate index is needed, especially when data resides on a secondary storage. Spatio-temporal indexing techniques abound since the late 1990s (extensions of R-tree or Quadtree variants, combined subdivisions in spatial and temporal domains, etc. [**44, 53**]). Throughout this work we focused on efficient in-memory pruning strategies, however, in Section 3.5 as part of our experimental observations, we provide a brief illustration about the benefits of using an existing index (TPR* tree [**74**]) for further improving the effects of the pruning. This, admittedly, is not a novel research or a contribution of this work, but it serves a two-fold purpose: (a) to demonstrate that our proposed approaches could further benefit by employing indexing; (b) to motivate further research for appropriate index structure.

3.3. Space Partitioning and Indexing

From the discussion of Section 3.2, and later observed in the experiments of Section 3.5, a significantly higher portion of $\vec{O}D$ events are pruned compared to $\vec{D}O$ events using our events pruning strategy. This is because an $\vec{O}D_{ij}$ event is only processed when both o_i and o_j is in the current MaxRS solution. On the other hand, a considerably larger number of objects are pruned while processing a $\vec{D}O$ event, with respect to processing an $\vec{O}D$ event. This occurs because the objects pruning in $\vec{D}O_{ij}$ only depends on o_i, o_j, and $CN(o_i, o_j)$.

Based on these observations, there are still room for improvements in: (1) Pruning more $\vec{DO}$ events; and (2) Objects pruning for $\vec{OD}$ events. In this section, we discuss additional indexing techniques that address both these issues. For the sake of simplicity, we'll discuss the examples for the *counting* version of the problem in the following discussions.

3.3.1. Intuition and Basic Idea

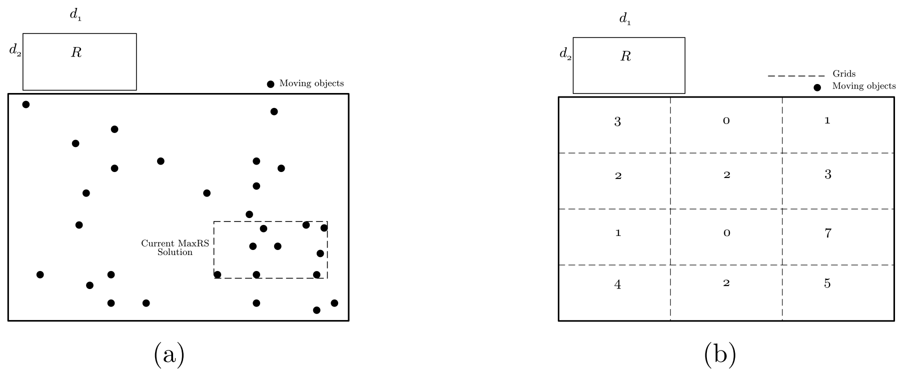


Figure 3.6. An example of grid partitioning: (a) R and a snapshot of moving objects (b) Count of objects for each grid cell at the given snapshot.

At first, let us consider the scenario when we are dealing with a snapshot (i.e., static version) of the moving objects databases – e.g., at a $\vec{DO}$ or $\vec{OD}$ event. Let us assume that (for now) the dimensions (i.e., d_1 and d_2) of the query rectangle R do not change, or change rarely. This is also quite common in practical scenarios, for example, suppose we have a camera with a given rectangular range, and we want to position the camera such that its coverage over a given set of moving objects (e.g., people walking around within a big hall, traffic moving along a road, etc.) is maximized. Then, the given rectangular range will not change unless a new camera model is used. In this setting, we propose an indexing scheme using grids of size R. We divide the whole space into $a \times b$ grids of size

R as shown in Figure 3.6. For each grid cell, we maintain the current count of objects (or, sum of the weights of the objects) within that grid. For an object, we also store the id/pointer of the current grid it is in. Given this information, we can compute the upper-bound of the possible *maximum score* of a rectangle R placed within that grid as follows:

Computing U-Bound. For any grid $G_{l,k}$, as size of each grid is equals to R itself, a rectangle R placed within the grid can intersect with at most 3 other neighboring grids simultaneously. In Fig. 3.7, we can see that at most 4 such overlapping cases (Q_1, Q_2, Q_3, Q_4) are possible. For each case, the maximum possible value of the overlapping R is equals to the sum of the current *count* of all the intersected neighboring grids (including the grid $G_{l,k}$). E.g., for the grid $G_{0,4}$ in Fig. 3.7:

$$Max_{Q1} = Count(G_{0,0}) + Count(G_{0,1}) + Count(G_{0,3}) + Count(G_{0,4})$$

We can similarly compute Max_{Q2}, Max_{Q3}, and Max_{Q4} by summing up counts of all the intersecting grids. Then, the *U-Bound* for any grid $G_{l,k}$ will be the maximum of these values. Thus, we can compute *U-Bound* for any grid $G_{l,k}$ as follows:

$$U\text{-}Bound(G_{l,k}) = max\{Max_{Q1}, Max_{Q2}, Max_{Q3}, Max_{Q4}\}$$

In Fig. 3.7, for grid $G_{0,4}$—$Max_{Q1} = 7$, $Max_{Q2} = 5$, $Max_{Q3} = 12$, and $Max_{Q4} = 6$. Thus, $U\text{-}Bound(G_{0,4}) = max\{7, 5, 12, 6\} = 12$.

Pruning. Using the techniques mentioned above, we can always compute *U-Bound* of a grid in $O(1)$ time after employing the proposed R-based grid partitioning, i.e., count of

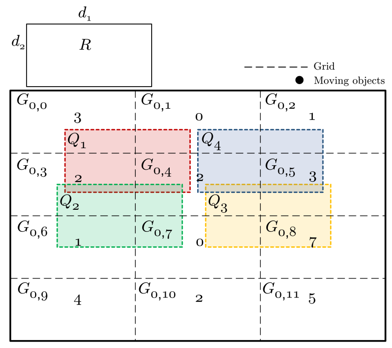


Figure 3.7. Computing *U-Bound* of a grid.

operations needed is constant – does not depend on the number of total grids or number of objects. First, to ensure more $\vec{DO}$ events are pruned, we derive the following lemma:

Lemma 6. *Consider the event $\vec{D}O_{i,j}$ for two objects o_i and o_j, occurring at time $t_{i,j}$. Let $l_{obj}^{(t_{i,j}-\delta)}$ (for some small δ) denote the Co-MaxRS solution just before $t_{i,j}$. Suppose, o_i and o_j belong to grids G_{l_i,k_i} and G_{l_j,k_j} respectively. In this setting, $l_{obj}^{(t_{i,j}-\delta)}$ remains a MaxRS if one of the following two inequalities holds:*

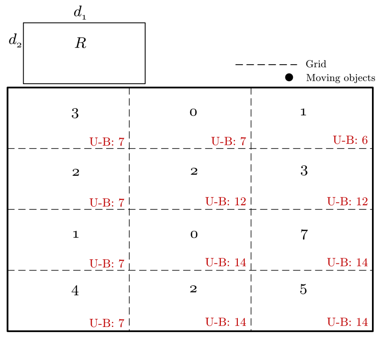
(1) U-Bound$(G_{l_i,k_i}) \leq score_{max}(t_{i,j} - \delta)$

(2) U-Bound$(G_{l_j,k_j}) \leq score_{max}(t_{i,j} - \delta)$

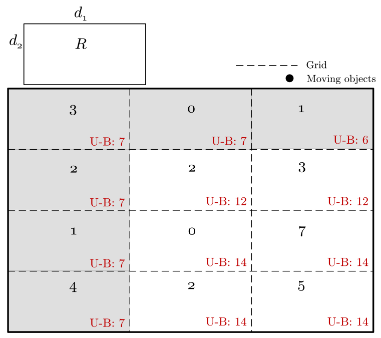
Subsequently, we propose the following lemma to improve the objects pruning strategy for $\vec{O}D$ events:

Lemma 7. *Consider the event $\vec{O}D_{i,j}$ for two objects o_i and o_j. Then, we can directly prune all objects of any grid $G_{l,k}$ from further computations, if U-Bound$(G_{l,k})$ $\leq score_{max} - min\{w_i, w_j\}$.*

Let us go back to the previous example of Fig. 3.6 and Fig. 3.7. In Fig. 3.8a, the count and *U-Bound* of all grids are shown. We can see that, $count_{max} = 9$ (see Fig. 3.6).



(a)



(b)

Figure 3.8. (a) Computing *U-Bound* of all the grids for example of Fig. 3.6 and Fig. 3.7 (b) Pruning grids and objects based on Lem. 7.

Thus, during an $\vec{O}D$ event, we can prune all the grids (and their objects) that have *U-Bound* ≤ 8 using Lem. 7. Fig. 3.8b shows the state of the grids after using the pruning scheme. All the gray-colored grids are pruned. In this example, half of the space and nearly 37% objects are pruned. This pruning would preclude the already in-place $\vec{O}D$ pruning strategies and make it even more effective.

3.3.2. Hierarchical Grid Indexing

Instead of restricting the effectiveness of the proposed partitioning to a particular size, we can employ a *Hierarchical Grid-Based Partitioning* over the bounding space $\mathbb{F}$ which will enable a larger set of R to be accommodated. Additionally, a hierarchical grid indexing will ensure more effective *U-Bound* computation and pruning method.

The hierarchical grid system to partition the space can be of any level l, where $l \geq 0$. An *l-level* hierarchical grid partitioning has $l+1$ levels in total, denoted as level $0, 1, \ldots, l$ respectively. The grid-size of level i is equals to the $\frac{1}{4}$-th of the grid of level $(i-1)$. Thus, a grid of level $(i-1)$ is composed of four level i grids. If the grid size for the level-0

Level 0
(a)
Level 0
Level 1
(b)

Figure 3.9. Examples of hierarchical grid partitioning of space: (a) 0-*level* (b) 1-*level*.

is $\frac{1}{4}\mathbb{F}$, then it is equivalent to a quad-tree. Examples of 0-*level* and 1-*level* hierarchical grid partitioning are shown in Fig. 3.9. In a hierarchical grid partitioning, we need to maintain count values only for the level l grids (i.e., smallest one), as count values of all other grid sizes of lower level (i.e., higher size) can be obtained from level l grid values. Additionally, we use $G_{l,k}$ to denote the k-th grid (starting from top-left corner) at level l (see Figure 3.10).

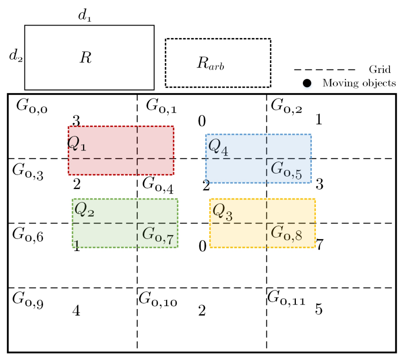


Figure 3.10. R_{arb} consumed by the grid size and maintaining the similar properties.

Enabling Arbitrary R**.** For any given query rectangle R_{arb}, we need to find the lowest numerical level for which the grid size is greater than R_{arb}. All of the findings and lemmas of Section 3.3.1 will still work for that level's grid size for the given R_{arb}. This is shown in Fig. 3.10. When using an *l-level* hierarchical grid indexing scheme, Lemma 6 and

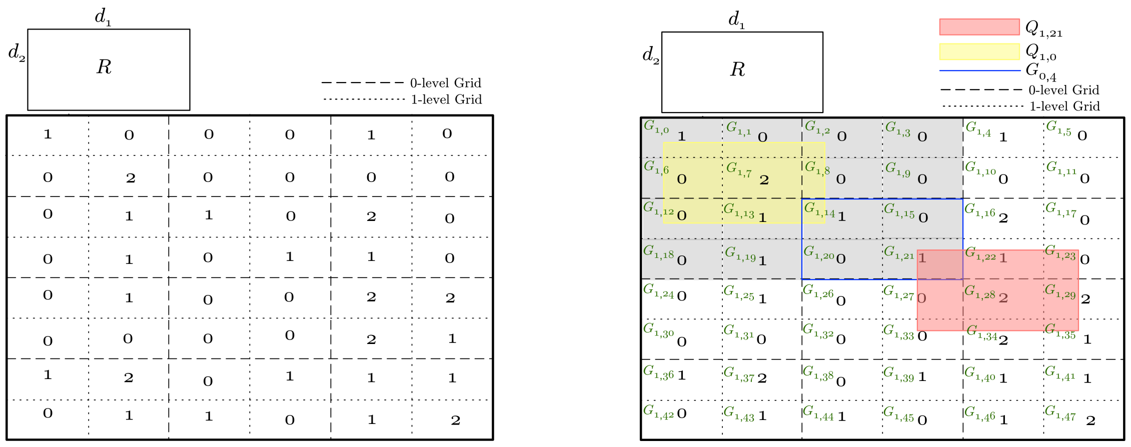
Lemma 7 can be applied even for different values of R_{arb}, as long as $R_l \leq R_{arb} \leq R_0$, where R_l and R_0 are the size of the grids at level l and 0 respectively.

Effective Pruning. With hierarchical grid indexing, we can compute *U-Bound* with higher granularity, as more level/depth of information is available, i.e., we can obtain lower *U-Bound* for grids. This, in turn, allows for higher number of grids and objects to be pruned in the indexing stage. Suppose, we apply a 1-*level* indexing for the example in Figure 3.6, where size of level 0 grids is equals to R. Then, the count of objects for each level 1 grid is shown in Figure 3.11a. Now, for any level 0 grid $G_{0,k}$, a rectangle R placed within the grid can intersect with at most 9 level 1 grid, e.g., $Q_{1,0}$ and $Q_{1,21}$ for $G_{0,4}$ in Figure 3.11b. We can uniquely identify each such possible set of $3 \times 3 = 9$ level 1 grids by the top-left grid of the set. For example, $Q_{1,0}$ represents the 3×3 level 1 grids starting from $G_{1,0}$. We can observe that at most 16 such overlapping cases are possible in Figure 3.11b, i.e., the grey colored $G_{1,k}$ grids, which represent the top-left grids for each of these 16 cases for $G_{0,4}$.

For each such case, the maximum possible value of the overlapping R equals to the sum of the current *count* of all the 9 level 1 grids. E.g., for the grid $G_{0,4}$ in Fig. 3.11b:

$$\begin{aligned} Max_{Q_{1,0}} = {} & Count(G_{1,0}) + Count(G_{1,1}) + Count(G_{1,2}) + Count(G_{1,6}) + Count(G_{1,7}) \\ & + Count(G_{1,8}) + Count(G_{1,12}) + Count(G_{1,13}) + Count(G_{1,14}) \end{aligned}$$

We can similarly compute the maximum possible value for all other cases. Then, the *U-Bound* for any level 0 grid $G_{0,k}$ will be the maximum of these 16 values. In Fig. 3.11b, for grid $G_{0,4}$, this value is 9 which occurs for $Q_{1,21}$ (the red rectangle). Thus, the *U-Bound*



(a) (b)

Figure 3.11. (a) Count of objects for level 1 grids (b) Computing *U-Bound* using level 1 grids.

for $G_{0,4}$ is 9, instead of 12 that we computed using only 0-*level* indexing in Section 3.3.1, i.e., reduced by 25%. Note that, the time-complexity of computing *U-Bound* for level 0 grids is still $O(1)$, although the constant value for the number of operations increases (i.e., now we have to consider 16 cases). Thus, although we can have better *U-Bound* values as we consider more level of grids, the computation time will increase as well, i.e., there is a trade-off between effectiveness of *U-Bound* values vs computation time.

3.3.3. Data Structures and Implementation Details

So far, we have discussed how the hierarchical grid partitioning will work over a snapshot of moving objects databases, e.g., at a $\vec{D}O$ or $\vec{O}D$ event. But, as the objects move, the count of objects within the grids change as well. We discuss the necessary data structures, algorithms, and implementation details to maintain the hierarchical grid partitioning for moving objects in the following.

Data Structures. Fig. 3.12 depicts the underlying data structures used to maintain hierarchical grid partitioning. A brief description of all the components are given below:

Object List (OL): We introduce an additional attribute for the list introduced in Section 3.2. For each object $O_i \in O$, we store a pointer to the current grid cell at level l. As an example (Fig. 3.12), O_2 is currently in grid $G_{l,2}$, meaning 2nd grid (starting from top-left) of level l. Note that, for brevity, we removed all the attributes we previously discussed about, and only show the new attribute in Fig. 3.12.

Hierarchical Grid List (GL): As illustrated in Fig. 3.12, we have a separate list for each grid-level. We denote *GL-i* as the list of grids at level i. Each element of a grid list stores relevant information about the grid, e.g., id, boundaries/edges, pointer to neighbors, and a current count of objects within the grid. Also, each grid element has a pointer to its parent grid at the immediately lower level, and four pointers to the children grids at the immediately upper level. Finally, only for grids at level l, we also store a list of pointers to the objects currently estimated to be inside the grid (cf. Fig. 3.12).

Kinetic Data Structures (KDS): We have already introduced the concept of KDS in Section 3.1 and Section 3.2. As the objects are moving, the count for each grid is changing as well. Note that, we just need to track changes for the grids at level l (i.e., the smallest grids), as the changes then can be propagated to the other levels via the parent pointers. There are two types of changes as before:

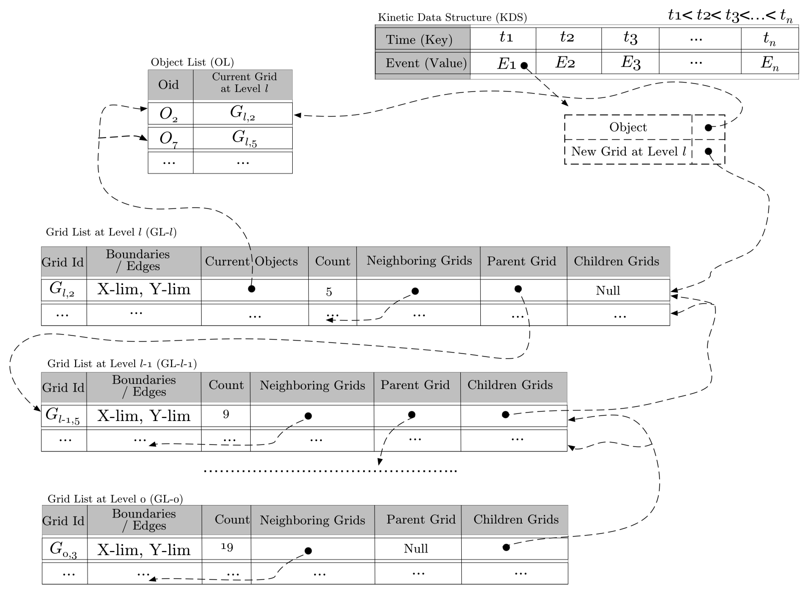

Kinetic Data Structure (KDS)
$t_1 < t_2 < t_3 < \ldots < t_n$
Time (Key)
Event (Value)
E_1 E_2 E_3 ... E_n
Object List (OL)
Oid
Current Grid at Level l
O_2 $G_{l,2}$
O_7 $G_{l,5}$
Object
New Grid at Level l
Grid List at Level l (GL-l)
Grid Id
Boundaries / Edges
Current Objects
Count
Neighboring Grids
Parent Grid
Children Grids
$G_{l,2}$ X-lim, Y-lim 5 Null
Grid List at Level l-1 (GL-l-1)
$G_{l-1,5}$ X-lim, Y-lim 9
Grid List at Level 0 (GL-0)
$G_{0,3}$ X-lim, Y-lim 19 Null

Figure 3.12. Hierarchical grid data structures.

- *Continuous Deformation:* The location of a moving object changes, but it still stays within the level l grid it was in.
- *Topological Change:* Due to the movement, an object moves into a new level l grid cell leaving the current one. This will result in changing the count of at least two level l grids.

Thus, the proper count of objects within the grid cells can be dynamically maintained by tracking only the topological changes (or, events) using the KDS framework. The event that is causing a topological change is the *grid-change event*. At each *grid-change event* of an object o_i at grid-level l, the appropriate counts for the old and new level l grids are updated at first. Subsequently, a new *grid-change event* is computed for o_i and the new grid. The updates of any level l grids can be propagated to lower level grids through parent pointers. An example is illustrated in Fig. 3.13. The locations of the object at time

t_1, t, and t_2 is shown in Fig. 3.13. Suppose, we are using 1-*level* grid partitioning, i.e., a *grid-change event* for the given object will occur when the object changes its current grid at level 1. The next *grid-change event* time can be computed by determining the intersection point(s) between the boundary of the grid and the linear movement function (time t in Fig. 3.13).

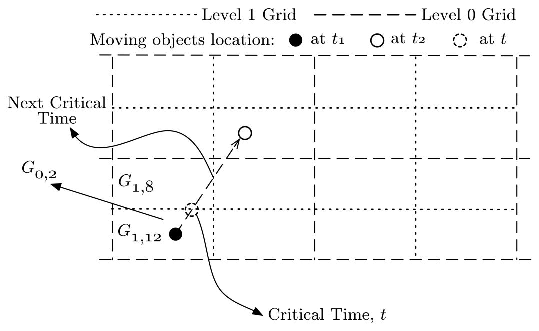


Figure 3.13. An example of *grid-change event* at time t.

Algorithm 4 HierarchicalEventProcess $(o_i, G_{l,k})$

```
1: $cgrid \leftarrow o_i.current\_grid$
2: $cgrid.count \leftarrow cgrid.count - 1$
3: $cgrid.current\_objects.Remove(o_i)$
4: **while** $cgrid = cgrid.parent$ is not NULL **do**
5:     $cgrid.count \leftarrow cgrid.count - 1$
6: **end while**
7: $G_{l,k}.count \leftarrow G_{l,k}.count + 1$
8: $G_{l,k}.current\_objects.Add(o_i)$
9: **while** $G_{l,k} = G_{l,k}.parent$ is not NULL **do**
10:     $G_{l,k}.count \leftarrow G_{l,k}.count + 1$
11: **end while**
12: Compute new event $E_{new}$ for $o_i$ and $G_{l,k}$
13: $KDS.Push(E_{new})$
14: $o_i.current\_grid \leftarrow G_{l,k}$
15: **return**
```

Algorithmic Details. In Algorithm 4, we present the grid-change event processing scheme for a given object o_i and its new grid $G_{l,k}$. At first, count and current objects list of the old grid cell is adjusted accordingly (lines 1-3). Then, this change is propagated to the lower levels via the parent pointer (lines 4-6). Similar updates are performed for the new grid $G_{l,k}$ from lines 7-11. Finally, the next grid change event is computed for the pair o_i and $G_{l,k}$, and pushed into KDS event queue in lines 12-13. Additionally, current grid cell of o_i is updated in line 14. Let us go back to the example of Fig. 3.13. At time t, count of $G_{1,12}$ is decreased by 1. This change is propagated to its parent $G_{0,2}$. Then, the count for $G_{1,8}$ is increased by 1 and this change is also propagated to its parent $G_{0,2}$. Thus, the count of $G_{0,2}$ remains unchanged. Finally, the next grid change event for $G_{1,2}$ is computed.

$\vec{OD}$ and $\vec{DO}$ Events Processing: We described the process of handling $\vec{OD}$ and $\vec{DO}$ events in Algorithm 3. We propose minor modifications to take advantage of the hierarchical grid partitioning in Algorithm 3. In lines 8-9, when pruning the $\vec{DO}$ events, we also check whether $U\text{-}Bound(o_i.current\_grid) \leq score_{max}$, or $U\text{-}Bound(o_j.current\_grid) \leq score_{max}$ – we can prune the $\vec{DO}$ event if either of the two statement is satisfied. Note that, this will be only be beneficial if we use an external database indexing scheme, i.e., no direct access to object list OL. Thus, it will be efficient to access the grid index first and do pruning there if possible. In line 20, we select the list of eligible objects OL' for an $\vec{OD}$ event – which is set to the list of all objects OL. Instead of selecting all the objects, now we need to first iterate through all the level 0 grids (i.e., size equals to R), and compute *U-Bound* for each one of them and check if Lemma 7 can be applied for

pruning. Eventually, the objects of the grids that are not pruned would be assigned to OL'.

Time-Complexity Analysis: The time-complexity of Algorithm 4 is $O(l)$. We will see later that (for both indexing and approximate solution), l is most of the times a small constant value. Thus, the time-complexity is practically $O(1)$. At any time, each object has exactly one grid-change event. Thus, size of KDS (considering only grid-change events) at any time instance is $O(n)$. Suppose, there are in total $a \times b$ level l grids. Then, for single-line trajectories, the maximum number of grid-change events is $(\sqrt{a^2 + b^2} - 1)$ – i.e., a diagonal straight line. Then, the total number of grid-change events in worst case scenario for the whole time duration $[t_0, t_{max}]$ is $n \times (\sqrt{a^2 + b^2} - 1)$. Finally, the first modification proposed in Algorithm 3 (i.e., lines 8-9) is $O(1)$ operation, while the second modification $O(|0\text{-}level|)$, where $|0\text{-}level|$ denotes the number of level 0 grids.

Discussion on Implementation. We presented a hierarchical grid-based partitioning of space in this section. To devise approximate solutions for Co-MaxRS problem, we will subsequently use the similar grid-based ideas as the base in the next section. We now discuss various possible methods of implementing the hierarchical grid-based indexing, and how we can incorporate it in our scheme – persistent or on-the-fly based on incoming queries. Grid-indexing [**61**], or similarly hierarchical grid-partitioning, can be implemented both as main-memory or external-memory based variants. We implemented the proposed hierarchical grid structure in main memory for our experiments. The reason is two-fold. (1) Many recent works have shown that main-memory indexes are usually necessary to provide high update and build performance [**72**] – which is paramount in dealing with moving objects databases; and (2) In our experiments, the maximum value of total number of objects is 50,000 — which can be stored in-memory.

On the other hand, as the time-interval of the Co-MaxRS query $[t_0, t_{max}]$ is usually large (e.g., some hours or days in our experiments), we build the hierarchical grid partitioning structure on-the-fly based on the given R for each query (i.e., level 0 grid size $= R$), and maintain the initially built structure throughout the query procedure via Algorithm 4. This is very reasonable as the hierarchical grid indexing build procedure is quite fast, and is subsumed by the overall query processing time. The cost of building an *l-level* grid partitioning is $O(l \times n)$, and as l is usually a small constant, the initial cost of building an *l-level* grid partitioning is practically $O(n)$. We can build the *l-level* grid partitioning on-the-fly in-between line 7 and 8 in Algorithm 2. In any case, if we have to process a bunch of queries for which time-interval is small (e.g., 10-15 minutes), we can always build an *l-level* grid partitioning once (with relatively large l value), and employ ideas of Section 3.3.2 to use the same data structure for different queries having different R values.

We note that, in extreme scenarios (e.g., millions of moving objects), the number of total objects may exceed the main memory storage capacity of servers – thus, external memory implementations and parallel processing of indexes would be necessary. Many works such as [**42, 45**] presented parallel processing techniques for R-trees and range queries. In a similar spirit [**90**] proposed a novel architecture named VegaGiStore, to enable efficient spatial query processing over big spatial data and concurrent access, via distributed indexing and map-reduce [**19**] technique. More recently, the SpatialHadoop [**22**] framework provided a library to perform map-reduce based parallel processing for many spatial operations, including grid indexing. We can use multiple grid indexing from SpatialHadoop simultaneously (each corresponding to a level of grids), and create an abstraction over them to implement a parallel hierarchical grid-indexing in a straightforward manner. We note, however, that Hadoop [**71**] as well as map reduce procedures may incur certain overheads – i.e., SpatialHadoop grid indexing will only be useful if there are an extremely large number of moving objects, and provided that there are a lot of resources (Hadoop nodes) readily available.

3.4. Approximate Co-MaxRS Solution

So far, we have proposed methods that compute the exact solution to the Co-MaxRS problem. The worst-case time-complexity of our exact solution for single-line trajectories is $O(n^3 \log n)$. We devised clever pruning strategies, and novel data structures to reduce this complexity significantly. On the other hand, in many practical scenarios, an exact solution is not necessary, rather than a fast but approximate solution with tight bounds of approximation ratio is desired. In this section, using the ideas established in Section 3.3, we introduce a fast approximate solution to the Co-MaxRS problem, and prove the tight upper-bound of the approximation ratio of our solution.

3.4.1. Basic Intuition

From the discussion of Section 3.3.3, for a Co-MaxRS query $(\mathbb{F}, O, R, [t_0, t_{max}])$, we can build an *l-level* hierarchical grid partitioning over $\mathbb{F}$ and O on-the-fly, and maintain the structure efficiently via Algorithm 4. Let *GL-l* represent the list of all level l grids, where size of level l grid is $\frac{R}{(2^l)^2}$ (see Figure 3.12), e.g., for GL-0 and $GL-1$, the grid size is R and $\frac{R}{4}$ respectively. Note that, GL-l does not represent hierarchical grids, rather it is just a *one-level* grid structure of particular unit grid-size $\frac{R}{(2^l)^2}$. As before, we use $G_{l,k}$ to denote the k-th grid of GL-l. At first, let us consider a basic GL-0 grid partitioning. As discussed in Section 3.3, we maintain the objects count for each grid cell via KDS. Additionally, for an approximation solution, we also keep track of the grid $G_{0,max}$, having the maximum count max_0 among all the grids. The intuition is: we can get a reasonable solution of $count_{max} = max_0$ to the Co-MaxRS problem, if we return the center of the $G_{0,max}$ (or, the objects within $G_{0,max}$) as our approximate Co-MaxRS solution at any time-instance. Note that, the approximate Co-MaxRS answer (i.e., the set of objects l_{obj}) can only change at a grid-change event, on two conditions: (1) Whenever a new $G_{0,max}$ is found with higher count value than the current solution; (2) The grid-change event occurs with respect to the current $G_{0,max}$. Note that, we do not need to track or process $\vec{D}O$ or $\vec{O}D$ events when computing approximate solution.

Approximation Ratio: We propose the following lemma regarding the approximation ratio for a GL-0 grid partitioning:

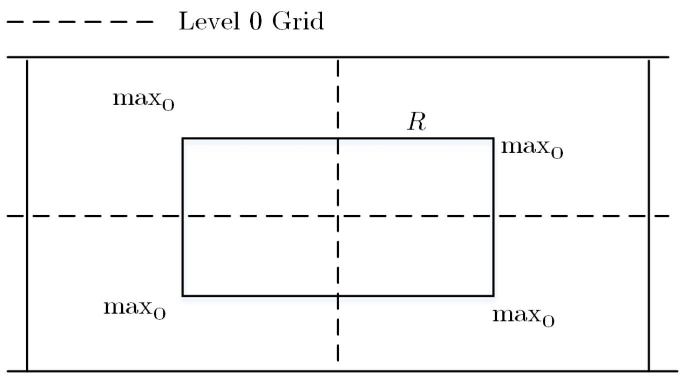


Figure 3.14. Approximation ratio for $l = 0$.

Lemma 8. *Using a GL-0 grid partitioning, the approximation ratio of the proposed approximation algorithm is 4.*

Proof. Suppose, max_0 is the maximum count among all the level 0 grids. Then, $count_{approx} = max_0$, where $count_{approx}$ is the computed approximate solution. On the other hand, we can place R within $\mathbb{F}$ such that it overlaps with 4 adjacent level 0 grids as shown in Fig. 3.14. In the worst case, all of the overlapped grids may have count value equals to max_0. Then the maximum possible value for exact MaxRS solution, $count_{max}$ $= 4 \times max_0$ (cf. Fig. 3.14). Thus, the approximation ratio for GL-0 is $= \frac{count_{max}}{count_{approx}} = \frac{4 \times max_0}{max_0} \Rightarrow 4$. $\square$

An example solution of this approach is shown in Figure 3.6 and Figure 3.7. Using this approach, the approximate solution would be the center of $G_{0,8}$ having count = 7. The exact MaxRS solution for this example, as shown in Figure 3.6a, is 9.

3.4.2. GL-l Based Solution

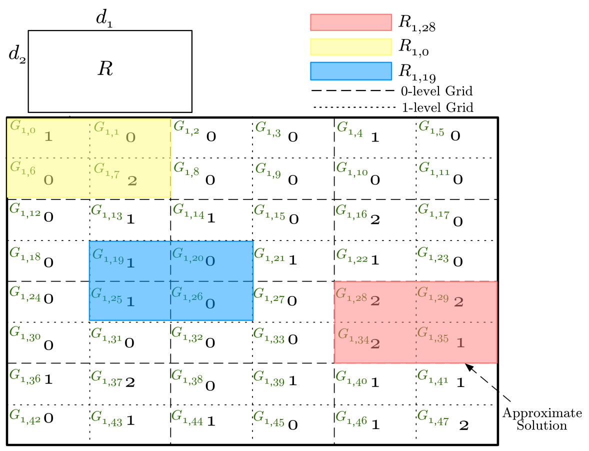


Figure 3.15. Approximation algorithm for GL-1 grid partitioning.

Using a GL-l grid-based partitioning (where $l \geq 0$), we can employ similar approximation algorithm inspired from GL-0 grid indexing. The advantage in this case is we have higher granularity information available. On the other hand, computing and maintaining the approximate solution will be costlier. The intuition is to work with the count values of the highest level grids, i.e., level l grids. For each grid $G_{l,k}$, we can place a rectangle R from its top-left corner point, encompassing 2^{l+1} level l grids. Such a rectangle placement is termed as an *incident rectangle* of any grid $G_{l,k}$, and we denote them as $R_{l,k}$. Then, the approximate solution will be the *incident rectangle* for which summed-up count value is maximum:

$$count_{approx} = max\{R_{l,k}.count\}_{\forall G_{l,k}}$$

Continuing with the example of Figure 3.6, suppose we apply GL-1 grid partitioning as shown in Figure 3.15. $R_{1,0}, R_{1,19}, R_{1,28}$ is shown in Figure 3.15, for level 1 grids $G_{1,0}, G_{1,19}, G_{1,28}$ respectively (i.e., yellow, blue, red). We can see that $R_{1,0}.count = 3$, and similarly, $R_{1,19}.count = 2$. Although, $R_{1,28}$ has the maximum count values among all incident rectangle placements, where $R_{1,28}.count = 7$. Thus, the center-point and objects of $R_{1,28}$ will be returned as the approximate solution. Although in this setting, the approximate solution is same as using a GL-0 grid-partitioning, in many cases a higher granularity (i.e., 4 times more incident rectangles) of information will lead to a better solution (see Section 3.5).

3.4.3. Algorithmic Details

In Algorithm 5, we present the detailed process for computing approximate Co-MaxRS for a given time period $[t_0, t_{max}]$ and query rectangle R. After initialization (lines 1-2), the *GL-l* grid partitioning is built on the fly in line 3. Let us assume that there are $a \times b$ level l grids in total for the given $\mathbb{F}$ and R, i.e., $|GL\text{-}l| = a \times b$. Note that, we only create a single grid partitioning (not hierarchical), and the time-complexity of this operation is $O(n)$, assuming $|GL\text{-}l| << n$. In line 4, we populate the KDS with the next grid-change events (or, line-change events – details omitted for brevity) for all the objects. This takes $O(n)$ time. Then we compute the approximate solution over the initial snapshot in line 5, by computing the count for all possible $R_{l,k}$ and keeping track of the maximum one throughout (i.e., $O(|GL\text{-}l|)$ time). Here, we track G_{opt} – the list of all the level l grids forming the approximate solution, i.e., $|G_{opt}| = (2^l)^2$. Lines 6-20 is similar to the Algorithm 2 – all the events in the KDS are processed in order of their time-value, and we maintain the approximate Co-MaxRS answer-set throughout. The main difference is in line 9, where the top event from the KDS is processed using the function *ApproxEventProcess* (elaborated in Algorithm 6). After checking whether a new solution has been returned by *ApproxEventProcess*, the answer-set (i.e., the time-interval and l_{obj}) is adjusted accordingly. We also update G_{opt} whenever a new solution is computed (cf. line 15). Note that, the condition check at line 10 in implementation actually takes constant time, which we detect via setting a boolean variable during approximate MaxRS computation.

The processing of a given grid-change event E_{cur} is shown in Algorithm 6. Initially, the object related to E_{cur} is retrieved, and then the count and objects list of the current grid cell is adjusted accordingly (lines 1-4). Subsequently, the new grid related to E_{cur} is set in line 5. Note that, E_{cur} is representing the critical time at which o_i is moving from $G_{l,cur}$ into the new grid $G_{l,new}$. Also, unlike algorithm 4, we are dealing with only a single

Algorithm 5 Approx-CoMaxRS (OL, R, t_0, t_{max}, l)

```
1: $KDS \leftarrow$ An empty priority queue of events (managed via $D_{KDS}$)
2: $\mathbb{A}_{Co\text{-}MaxRS} \leftarrow$ An empty list of approximate answers
3: $GL\text{-}l \leftarrow$ BuildGridIndex ($OL$, $R$, $l$)
4: Compute next grid-change event $E_{next}$, $\forall o_i \in OL$ and push to $KDS$
5: $(loc_{opt}, count_{approx}, l_{obj}, G_{opt}) \leftarrow$ R_Approx_MaxRS ($GL\text{-}l$)
6: $t^{new}_{start} \leftarrow t_0$
7: while $KDS$ not EMPTY do
8:     $E_{cur} \leftarrow KDS.Pop()$
9:     $(l'_{obj}, count_{approx}, G'_{opt}) \leftarrow$ ApproxEventProcess ($E_{cur}, KDS, l_{obj}$, $count_{approx}$, $G_{opt}$)
10:    if $l_{obj} \neq l'_{obj}$ then
11:        $t^{new}_{end} \leftarrow t_i$
12:        $\mathbb{A}_{Co\text{-}MaxRS}.Add(l_{obj}, [t^{new}_{start}, t^{new}_{end}))$
13:        $t^{new}_{start} \leftarrow t_i$
14:        $l_{obj} \leftarrow l'_{obj}$
15:        $G_{opt} \leftarrow G'_{opt}$
16:    end if
17: end while
18: $t^{new}_{end} \leftarrow t_{max}$
19: $\mathbb{A}_{Co\text{-}MaxRS}.Add(l_{obj}, [t^{new}_{start}, t^{new}_{end}))$
20: return $\mathbb{A}_{Co\text{-}MaxRS}$
```

level grid partitioning, and thus there is no need to propagate updates to parent or child grids. In lines 6-7, the count and objects list of the new grid cell is adjusted accordingly. Current grid cell of o_i is updated in line 8. From lines 9-12, we check if $G_{l,cur}$ is in G_{opt} – which means current $count_{max}$ is decreased and a new solution may be possible. Only in this case, we recompute the approximate solution by going over all the level l grids (i.e., $R_{l,k}$). Otherwise, we only need to consider the rectangle placements which overlap $G_{l,new}$. This is done in lines 13-21. At first, the list of all $R_{l,k}$ overlapping $G_{l,new}$ (i.e., $Rlist_{new}$) is computed in line 13. Here, $|Rlist_{new}| = (2^l)^2$. For all such rectangle placements, their cumulative count is computed and checked if its larger than the current approximate solution – in which case, a new solution is found and related variables are updated accordingly (cf. lines 15-19). The cost-sensitive operations for Algorithm 6 is line 10, and lines 13-21. Line 10 takes $O(|GL\text{-}l|)$ time in worst-case, while lines 13-21 take $O((2^l)^2)$ time. As $|GL\text{-}l| > (2^l)^2$ (for small values of l), the worst-case time complexity for Algorithm 6 is $O(|GL\text{-}l|)$.

Algorithm 6 ApproxEventProcess (E_{cur}, KDS, l_{obj}, $count_{approx}$, G_{opt})

```
1:  $o_i \leftarrow E_{cur}.object$
2:  $G_{l,cur} \leftarrow o_i.current\_grid$
3:  $G_{l,cur}.count \leftarrow G_{l,cur}.count - 1$
4:  $G_{l,cur}.current\_objects.Remove(o_i)$
5:  $G_{l,new} \leftarrow E_{cur}.grid$
6:  $G_{l,new}.count \leftarrow G_{l,k}.count + 1$
7:  $G_{l,new}.current\_objects.Add(o_i)$
8:  $o_i.current\_grid \leftarrow G_{l,new}$
9:  if $G_{l,cur} \in G_{opt}$ then
10:     $(loc'_{opt}, count'_{approx}, l'_{obj}, G'_{opt}) \leftarrow$ R_Approx_MaxRS ($GL\text{-}l$)
11:     return ($l'_{obj}$, $count'_{approx}$, $G'_{opt}$)
12: end if
13: $Rlist_{new} \leftarrow \bigcup_{k=1}^{a \times b} R_{l,k},\ if\ G_{l,new} \cap R_{l,k} \neq \emptyset$
14: for all $R_{l,k}$ in $Rlist_{new}$ do
15:     Compute $R_{l,k}.count$
16:     if $R_{l,k}.count > count_{approx}$ then
17:         $count_{approx} \leftarrow R_{l,k}.count$
18:         $G_{opt} \leftarrow \bigcup_{i} G_{l,i},\ if\ R_{l,k} \cap G_{l,i} \neq \emptyset$
19:         Compute $l_{obj}$ from $G_{opt}$
20:     end if
21: end for
22: return ($l_{obj}$, $count_{max}$, $G_{opt}$)
```

Approx-CoMaxRS Processing Time: The time-complexity to maintain this approximate solution is very fast compared to the exact solution. In KDS, we will only have grid-change events and line-change events, i.e., we do not have to consider the $\vec{D}O$ and $\vec{O}D$ events. For a single-line trajectory, the maximum number of grid-change events is

$(\sqrt{a^2+b^2}-1)$ – i.e., a diagonal straight line. Secondly, during each grid-change event, the processing time is $O(|GL\text{-}l|)$, or $O(a \times b)$ in worst-case. Thus, in addition to the $O(n)$ initialization cost (i.e., lines 3 and 4), the additional time-complexity for Algorithm 5 is $O(a \times b \times (\sqrt{a^2+b^2}-1))$. As we increase the value of l, the value of a and b increases as well. Thus, there is a trade-off between quality of approximation vs computational cost.

3.4.4. Approximation Ratio

Let us denote max_l as the approximate solution obtained by using Algorithm 5 (Approx-Co-MaxRS) over a GL-l grid partitioning. Then, we propose the following lemma:

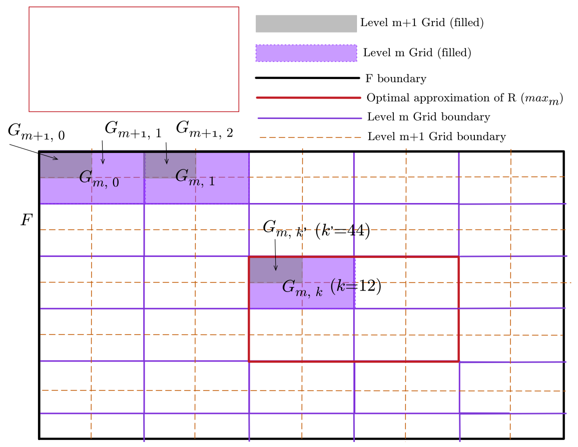


Figure 3.16. Proof of Lemma 9.

Lemma 9. *For any GL-m grid partitioning, $max_{m+1} \geq max_m$, where $m \geq 0$.*

Proof. We know that, each level m grid is composed of exactly 4 level $m+1$ grids. For example, $G_{m,0}$ consists of $G_{m+1,0}$, $G_{m+1,1}$, and two other level $m+1$ grids (cf. Figure 3.16). Thus, total number of grids in GL-$m+1$ = 4 × total number of grids in GL-m. Subsequently, there will be 4 times more incident rectangle placements (i.e., $R_{l,k}$) to select from. Also, for each $R_{m,k}$ in GL-(m), there is a similar rectangle placement $R_{m+1,k'}$ in GL-$(m+1)$. Suppose, $R_{m,k}$ has the maximum count among all the incident rectangles in GL-m, i.e., max_m. In Figure 3.16, $R_{m,k}$ is the red-stroked rectangle. For simplicity, let us follow the grid numbering scheme of Figure 3.16 for any level grid-partitioning – top-left cell is 0, and then count is increased horizontally. Then, $R_{m,k} = R_{m+1,k'}$, where $k' = (4 \times \beta \times \lfloor \frac{k}{\beta} \rfloor) + ((k \mod \beta) \times 2)$. Here, β is the total number of level m grids necessary to horizontally cover $\mathbb{F}$. This is shown in Figure 3.16 (β=5, $k = 12$, $k' = 44$). Thus, count of $R_{m+1,k'}$ will be max_m as well. As given any maximal incident rectangle placement $R_{m,k}$, there will always be another incident rectangle $R_{m+1,k}$ with same count value max_m, the approximate solution for GL-$m+1$ will be at least max_m. Thus, $max_{m+1} \geq max_m$. □

Using Lemma 8 and 9, we propose the following theorem regarding the approximation ratio for a GL-l grid partitioning:

Theorem 1. *For a given GL-l grid partitioning, where $l \geq 0$, the approximation ratio of Approx-CoMaxRS algorithm is* 4.

Proof. We will prove this claim by induction.

Step 1: Let, $l = 0$. Then, we will have a GL-0 grid partitioning. From Lemma 8, the approximation ratio for GL-0 grids is 4.

Step 2: Suppose, for $l = m$, Theorem 1 is satisfied, i.e., in the worst case scenario, $count_{max} = 4 * max_m$, where max_m is the approximate solution returned via the GL-m grid partitioning.

Step 3: Now, using *Step 2*, we have to show that Theorem 1 holds for $l = m+1$ as well. From Lemma 9, we know that $max_{m+1} \geq max_m$. Thus, from Step 2, approximation

ratio for $l = m + 1$ is : $\frac{count_{max}}{count_{approx}} = \frac{4 \times max_m}{max_{m+1}} \leq 4$. As the approximation ratio cannot be higher than 4 for GL-$m+1$, to prove our theorem it is sufficient to show that there is a configuration for which the approximation ratio for $l = m+1$ is 4. In case of GL-$(m+1)$ grids, a rectangle R is composed of $(2^{m+1})^2$ level $m+1$ grids, and we can place R such that it overlaps with $(2^{m+1}+1)^2$ level $m+1$ grids. An example of such configuration is shown Figure 3.17, where a placement of R overlaps with $(2^{m+1}+1)^2$ level $m+1$ grids. In the worst-case scenario, suppose count of all $G_{m+1,k}$ is 0, except for the corner cells, e.g., G_{m+1,k^1}, G_{m+1,k^2}, G_{m+1,k^3}, and G_{m+1,k^4} in Figure 3.17. Suppose, count of the corner cells are max_m. Then, $count_{approx}$ computed via GL-$(m+1)$ will be max_m – i.e., any of R_{m+1,k^1}, R_{m+1,k^2}, R_{m+1,k^3}, and R_{m+1,k^4} can be the solution. Note that, such placing also does not increase the approximate solution returned by GL-m (see the purple level m grids in Figure 3.16). From Step 2, we already have, $count_{max} = 4 * max_m$. Subsequently, the approximation ratio via GL-$(m+1)$ is, $\frac{count_{max}}{count_{approx}} = \frac{4 \times max_m}{max_m} \Rightarrow 4$.
Thus, Theorem 1 is proved by induction.

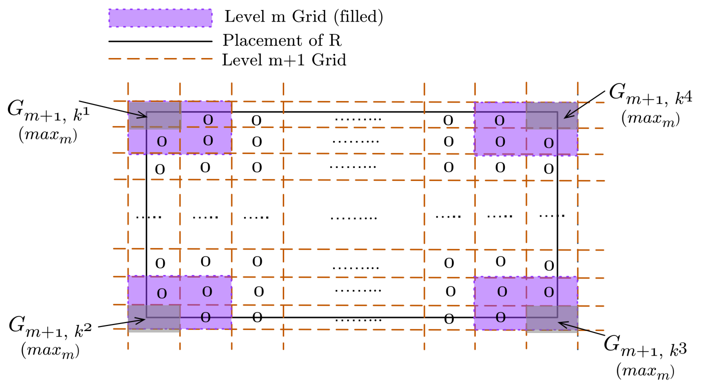


Figure 3.17. Lower bound of approximation ratio for $l = m + 1$ is 4.

$\square$

We note that, the ratio between the exact and approximate Co-MaxRS solution in practice is significantly less than the theoretical approximation ratio proved here (details in Section 3.5).

3.5. Experimental Observations

3.5.1. Setup

labelexpsetup **Datasets**: We used two real-world datasets and another synthetic one during our experiments. The first real-world dataset we used is the bicycle GPS (BIKE-dataset) collected by the researchers from University of Minnesota [**29**], containing 819 trajectories from 49 different participant bikers, and 128,083 GPS points. The second one is obtained from [**88**] (MS-dataset), which contains GPS-tracks from 182 users in a period of over five years collected by researchers at Microsoft with 17,621 trajectories in total, covering 1,292,951 km and over 50,176 hours (with GPS samples every 1-5 seconds). To demonstrate the scalability of our approach, we also used a large synthetic dataset

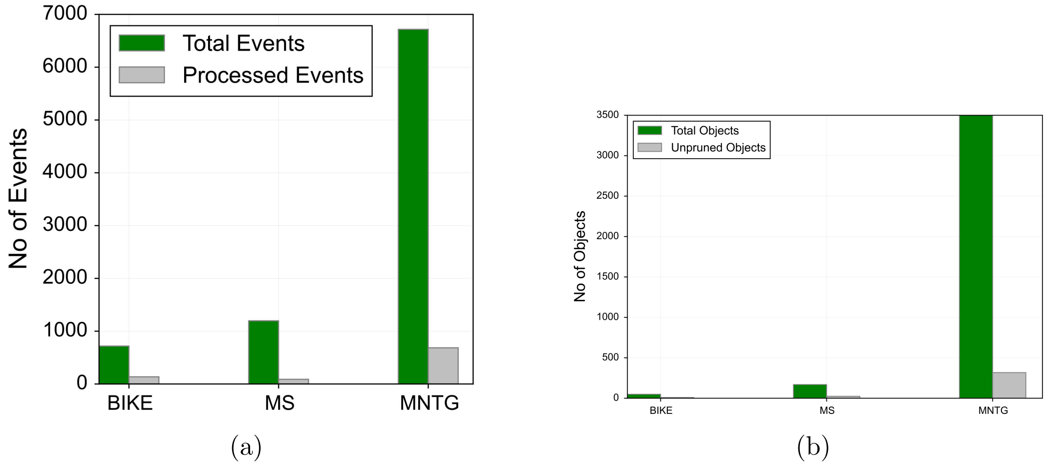


Figure 3.18. (a) Events Pruning (b) Objects Pruning.

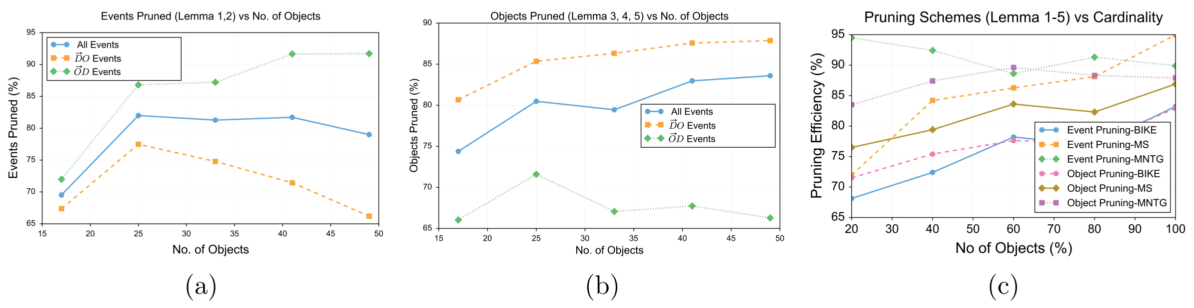


Figure 3.19. Impact of cardinality on the pruning schemes: (a) Different events pruning (BIKE-dataset) (b) Objects pruning (BIKE-dataset) (c) Overall objects and events pruning (all datasets).

(MNTG-dataset) generated using Minnesota Web-based Traffic Generator [**52**]. The generated MNTG-dataset consists of 5000 objects, and 50000 trajectories with 400 points each, where we set the option that objects are not constrained by the underlying network. For every object in the synthetic dataset, we generated its weight uniformly in the range from 1 to 50, while weights in Bike-dataset and MS-dataset (real-world datasets) were set to 1.

For each of the dataset used in the experiments, we considered one trajectory per object during a run and we averaged over all the runs to get representative-observations. The default values of the number of objects for BIKE, MS, and MNTG dataset are 49, 169, and 5000 respectively. The query time is set to the whole time-period (lifetime of trajectories) during a particular run for each respective dataset, and the base value of range area (R) for each of the BIKE, MS, and MNTG dataset is 4000000, 1000000, and 4000000 m^2 respectively.

Implementations: We implemented all the algorithms in Python 2.7, aided by powerful libraries, e.g., Scipy, Matplotlib, Numpy, Rtree, etc. We conducted all the experiments on a machine running OS X High Sierra, and equipped with Intel Core i7 Quad-Core 3.7 GHz CPU and 16GB memory. As no prior works exist that directly deal with the Co-MaxRS problem, we use Algorithm 1 as our baseline for comparison. In addition to the Algorithms 1, 2 and 3, we have two additional implementations[3]. Couple of things to note here: (1) To demonstrate the benefits of our pruning schemes, we tested them against a trivial approximate-solution to Co-MaxRS: one that would periodically re-evaluate the query throughout its time-interval of interest. In other words, MaxRS is re-computed at each $t+\delta$, i.e., δ is a fixed time-period (default δ=5s); (2) We also did an experiment using TPR-tree – a popular index structure for moving objects trajecotries [**53**], to evaluate the performance of our proposed hierarchical grid indexing scheme.

Results: We present our observed experimental results and analysis as following: (1) First, we demonstrate the effectiveness of our proposed pruning techniques and the exact Co-MaxRS algorithm; (2) Subsequently, we focus on the performance vs accuracy trade-off for the Approx-Co-MaxRS algorithm; and (3) Finally, we show the speed-up obtained by using our offered hierarchical grid index structure. We vary the values of a set of related parameters, i.e., grid size R, grid-level l, time-period δ, and number of objects $|O|$ to obtain a thorough evaluation. We use the experiments on the MNTG-dataset to demonstrate scalability of our solutions.

3.5.2. Co-MaxRS

Experimental results related to the Co-MaxRS algorithm are as follows.

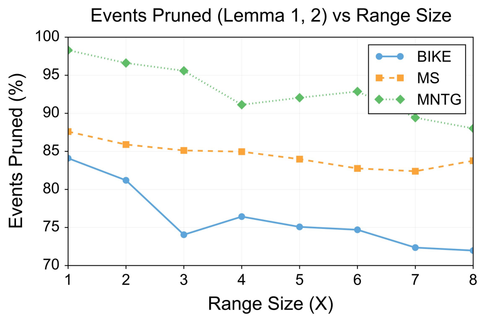
Performance of Pruning Strategies: Our first observations are shown in Figure 3.18a

[3] We note that all the datasets and the source code of the implementation are publicly available at http://www.eecs.northwestern.edu/∼mmh683.

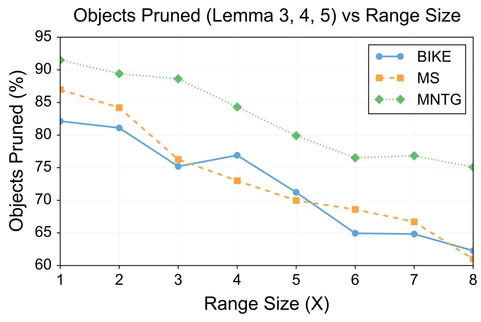
and they demonstrate the effectiveness of our events pruning strategy over both the real and synthetic datasets. The most amount of pruning is obtained in MS-dataset, while the other two datasets also show more than 80% pruning. Note that, the number of actual recomputation-events are well below the worst-case theoretical upper-bound, e.g., only 103 events are processed for 49 objects (trajectories) running for an hour in Bike-dataset. Similar results are obtained for the objects pruning scheme, as demonstrated in Figure 3.18b – indicating that the pruning schemes perform nearly equally well in all three datasets.

Impact of Cardinality: Figure 3.19 illustrates the impact of the cardinality on the effectiveness of our pruning methods. In Figure 3.19a, from the experiment done on the BIKE-dataset, we can deduce an interesting relation: as the datasize increases, more $\vec{OD}$ kind of events are pruned, whereas (cf. Figure 3.19b), objects pruning slightly decreases for $\vec{OD}$ as the datasize increases. On the other hand, $\vec{DO}$ events exhibit completely opposite behavior. This, in a sense, neutralizes the overall impact of the increase in cardinality for our pruning scheme. Figure 3.19c demonstrates the effect of increasing the cardinality of objects on the pruning schemes for all the dataset – hence, the label on the X-axis indicates the percentage of all the objects for the respective datasets.

Influence of Range Size: This experiment was designed to observe the effect of different range sizes, i.e., the area of $R - d_1 \times d_2$ over the pruning strategies. As shown in Figure 3.20a, increasing range area (the values on X-axis indicate multiples of the base-size for each dataset) results in fewer portion of events pruned. This occurs because as the area of R grows, there are more overlapping dual rectangles among the moving objects. Similarly, the growing rectangle size had adverse effects on the objects pruning scheme



(a)



(b)

Figure 3.20. (a) Events pruning strategy; (b) Objects pruning strategy against varying range sizes.

as well (cf. Figure 3.20b). We note, though, that even with quite large values of R (e.g., 500000 m^2) we have more than 60% of pruning through our proposed methods.

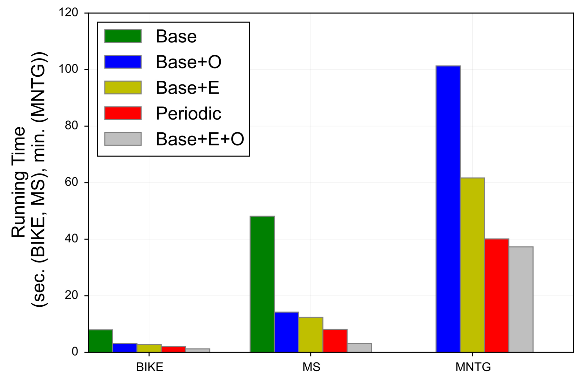
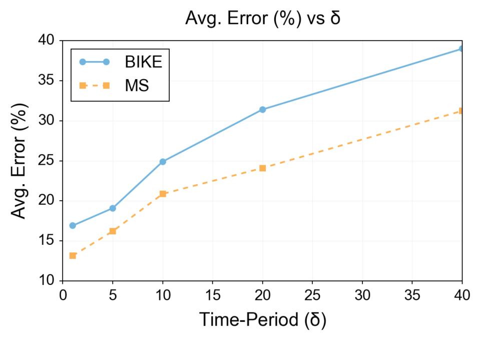
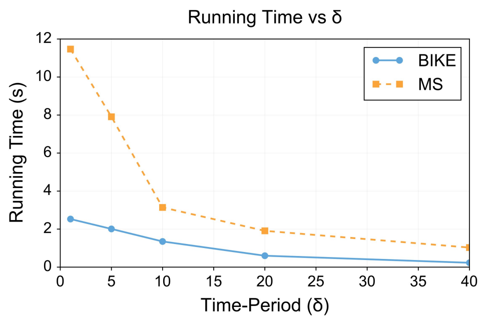


Figure 3.21. Running-time in different datasets.

Running Time Comparison: We ran the algorithms over the three datasets and the result is shown in Figure 3.21. This is the first experiment in which we also report observations regarding the periodical processing of the MaxRS – and it serves the purpose to provide a complementary illustration of the benefits of our methodologies. Namely, even if one is willing to accept an error in the result and perform only periodic snapshot MaxRS, our pruning techniques are still more efficient, while ensuring correct/complete answer set. The *Base*, *(Base+O)*, *(Base+E)*, *(Base+E+O)*, and *Periodic* in Figure 3.21 denote the base Co-MaxRS, base + objects pruning, base + events pruning, base + both events and objects pruning, and periodical processing of MaxRS (δ=5s), respectively. In case of MNTG-dataset, the average running time (for a set of trajectories) is shown in minutes, while for the other two datasets the unit it is shown in seconds. We omitted the average running time for the base algorithm over MNTG-dataset in Figure 3.21 which is more than 10 hours (to avoid skewing the graph). The base Co-MaxRS is the slowest among these algorithms, as it recomputes MaxRS at each event. The effect of both events and objects pruning schemes on running time is prominent, although events pruning exhibits a bigger impact individually (preventing unnecessary recomputations). When both pruning strategies are applied together, the algorithm speeds-up significantly – almost 6-15 times faster than the base algorithm over all the datasets – making it the fastest among all the evaluated algorithms.



(a)



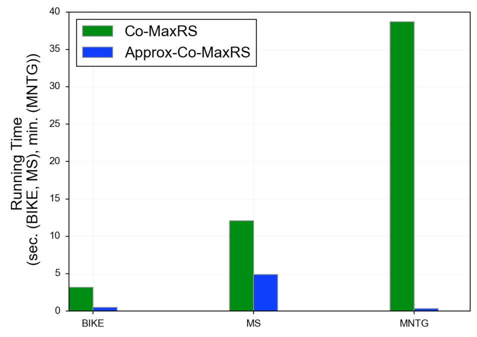
(b)

Figure 3.22. Impact of δ on (a) Error (b) Running Time of periodic-MaxRS.

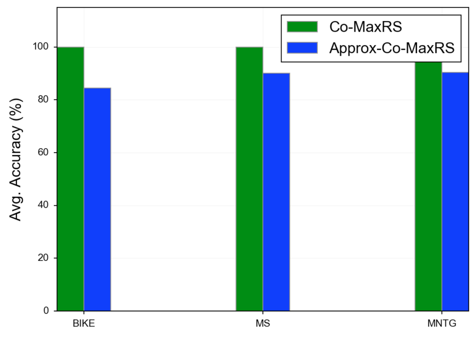
Periodical Processing: The last observations illustrate the errors induced by periodical processing of MaxRS (periodic-MaxRS) to approximate Co-MaxRS. Note that we exclude performing periodic-MaxRS related experiments on the large synthetic dataset (MNTG-dataset) as the correctness, rather than scalability, is a concern. In Figure 3.22, the impact of (δ) is illustrated both on running time and correctness. As δ increases the error in the approximation increases as well. Even for a small δ (e.g., $1s$), the respective error is still around 8-14% (cf. Figure 3.22a). Complementary to this, in Figure 3.22b, we see that as δ decreases, the running time increases too. For both Bike-dataset and MS-dataset, for small δ values ($\leq 5s$), average processing time is much longer than our proposed algorithm ($Base{+}E{+}O$) and yet it contains errors.

3.5.3. Approx-Co-MaxRS

Experimental results related to the approximate solution of Co-MaxRS are analyzed in this section. The default value of grid-level l is set to 1, i.e., if not mentioned otherwise,



(a)

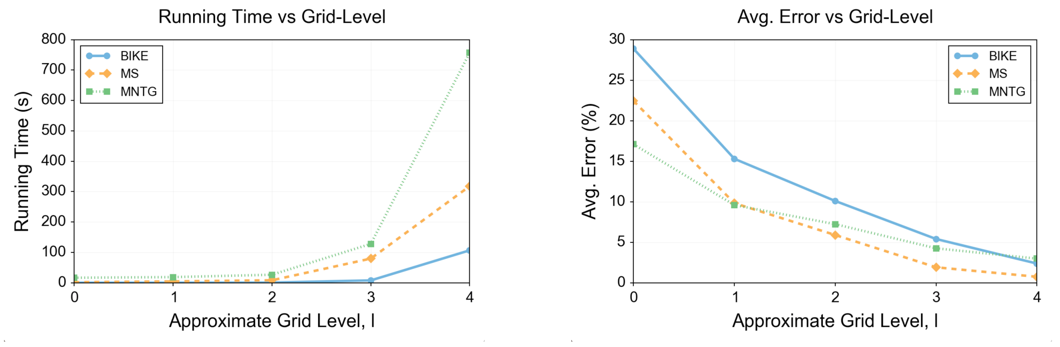


(b)

Figure 3.23. (a) Running Time (b) Accuracy (of approximate answers).

GL-1 grid partitioning is employed to compute approximate answers.

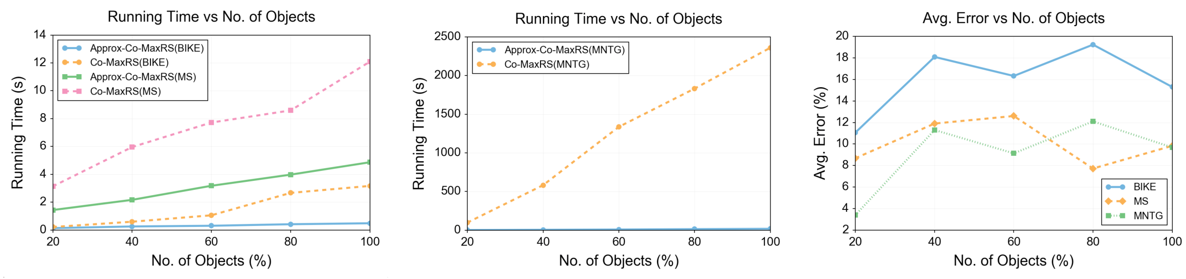
Performance – Running Time vs Accuracy: Our first observations are shown in Figure 3.23a and they demonstrate the running time of Approx-Co-MaxRS compared to the exact algorithm. Approx-Co-MaxRS is significantly faster over both the real and synthetic datasets. The difference in processing time is most skewed in MNTG-dataset, where Approx-Co-MaxRS takes only 18.9 seconds to complete, compared to 38.7 minutes to obtain the exact solution. On the other hand, even with this fast running time, the Approx-Co-MaxRS generates answers with relatively low error as shown in Figure 3.23b. For both MS and MNTG-dataset, the approximate solutions achieve more than 90% avg. accuracy – which amounts to an avg. approximation ratio of 1.1, significantly lower than the theoretical bound of 4. The results in Figure 3.23 show that Approx-Co-MaxRS maintains relatively accurate answers with remarkably faster response time.



(a) (b)

Figure 3.24. Impact of l on (a) Running Time (b) Error of Approx-Co-MaxRS.

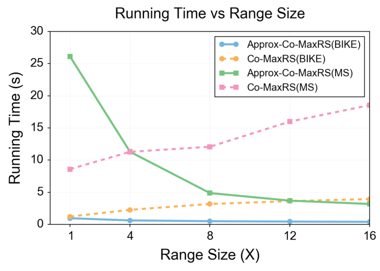
Varying Grid Level l: Figure 3.24 illustrates the impact of the grid-level l on the performance of our approximate solution. An increased value of l results in higher processing time (cf. Figure 3.24a) and lower average error (cf. Figure 3.24b). So, this direct trade-off between performance and accuracy has to be considered when choosing an appropriate value of l for the Approx-Co-MaxRS algorithm. The phenomena of higher processing time is particularly evident for $l = 3$ and 4 (processing time worse than exact Co-MaxRS for BIKE and MS dataset), while the accuracy gets a hit when l=0. Based on the experiments over the three datasets, an l value of 1 and 2 comes across as reasonable choices.



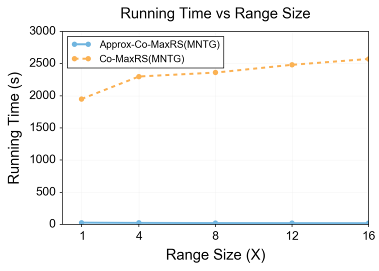
(a) (b) (c)

Figure 3.25. Impact of cardinality on: (a) Running Time (MS and BIKE-dataset) (b) Running Time (MNTG-dataset) (c) Error (all datasets) of Approx-Co-MaxRS.

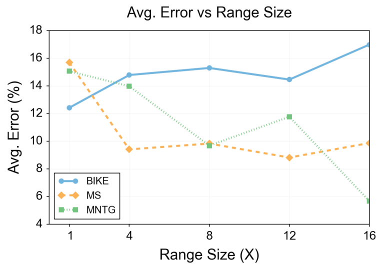
Impact of Cardinality: This experiment was designed to scrutinize the effect of cardinality on the running time and accuracy (i.e., avg. error) of the Approx-Co-MaxRS algorithm. We observe results from various datasets in Figure 3.25 – hence, the label on the X-axis indicates the percentage of all the objects for the respective datasets. As the datasize increases, processing time increases as well for both Approx-Co-MaxRS and Co-MaxRS algorithms (see Figure 3.25a and 3.25b). Moreover, an interesting observation is that the difference between the processing time of Approx-Co-MaxRS and Co-MaxRS also increases with the datasize – i.e., the advantage of using Approx-Co-MaxRS for a quicker response is more prominent for larger datasets (cf. see Figure 3.25b). The avg. error of obtained approximate answers over all datasets do not fluctuate much with varying datasize as shown in Figure 3.25c.



(a)



(b)



(c)

Figure 3.26. Impact of range size on: (a) Running Time (MS and BIKE-dataset) (b) Running Time (MNTG-dataset) (c) Error (all datasets) of Approx-Co-MaxRS.

Influence of Range Size: Figure 3.26 demonstrates the influence of range size R on the performance of Approx-Co-MaxRS. In contrast to the Co-MaxRS algorithm (i.e., pruning performance gets worse with larger R), the running-time of Approx-Co-MaxRS decreases as we increase the values of R over all three datasets (cf. Figure 3.26a and Figure 3.26b). This is due to the fact the total number of grids is fewer for larger R, which in turn results in fewer grid-change event occurrences and lower maintenance overhead for the algorithm. Note that, the values on X-axis indicate multiples of the base-size for each dataset in Figure 3.26. On the other hand, Figure 3.26c shows that avg. error is not that affected by the change in R. From the experimental results, we can conclude that the avg. accuracy of Approx-Co-MaxRS depends mostly on the grid-level l.

Avoiding Full Recomputations: In Algorithm 6, for a grid-change event E_{cur} where an object is moving from $G_{l,cur}$ to a new grid $G_{l,new}$, we only perform full recomputation

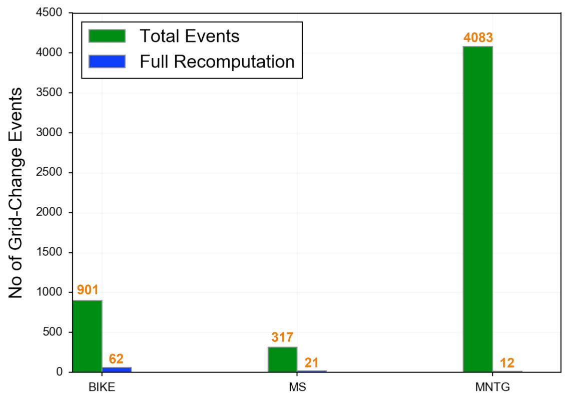


Figure 3.27. Pruning (avoiding full recomputation) in grid-change events.

of apparox-maxrs if $G_{l,cur}$ is part of the current solution (line 9-12 in Algorithm 6). In all other cases, we avoid this relatively expensive full-recomputation ($O(n * 2^{2l})$) and only check $R_{l,k}$'s involving the new grid $G_{l,new}$. The benefits of using this idea in Algorithm 6 is revealed in Figure 3.27. For BIKE and MS dataset, full recomputation is necessary for only around 6% of the grid-change events, whereas this number goes down to only 0.3% for the MNTG-dataset.

3.5.4. Hierarchical Grid-Based Indexing

Experimental results related to the hierarchical grid-based indexing scheme (denoted as Co-MaxRS-HGrid) are presented below. We implement in-memory index structures for our experiments. As indexing is effective mostly for large dataset, we conduct experiments only over MNTG-dataset for this part. Similar to Approx-Co-MaxRS experiments, the default value of grid-level l is set to 1.

Performance Comparison: Our first observations are shown in Figure 3.28, where we compare the running time between Co-MaxRS implementation: (1) without any indexing;

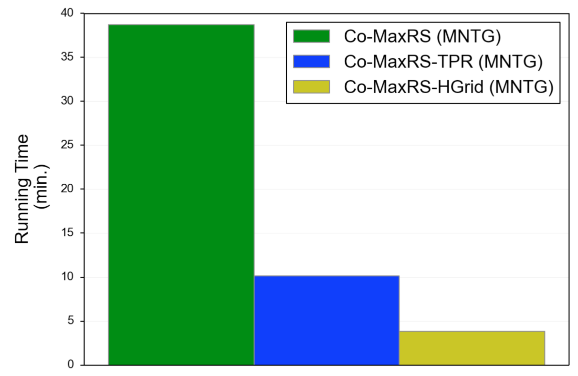
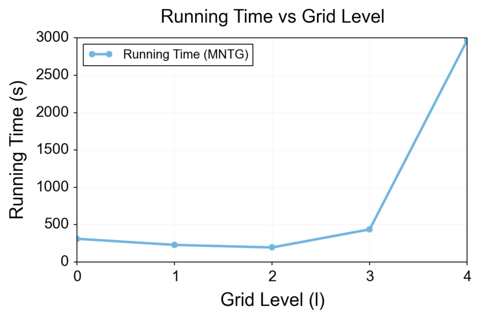


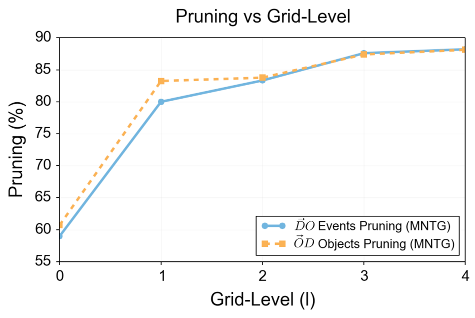
Figure 3.28. Running time comparison between Co-MaxRS, Co-MaxRS-TPR, and Co-MaxRS-HGrid.

(2) with TPR-tree indexing; and(3) with our proposed hierarchical grid-indexing. The results demonstrate that both indexing schemes reduce the processing time significantly, by speeding-up various range queries involved in the Co-MaxRS algorithm (e.g., find the set of overlapping rectangles for a given rectangle at each line-change event). Although, the performance of Co-MaxRS-HGrid is 3 times faster than the TPR-tree version (cf. Figure 3.28), attributed to following reasons: (1) The efficient $\vec{DO}$ events and $\vec{OD}$ objects pruning obtained via Co-MaxRS-HGrid; (2) The relatively low build and maintenance cost of Co-MaxRS-HGrid – e.g., the time-complexity of Algorithm 4 is $O(l)$, or $O(1)$ for the experiments; and (3) The comparatively high index building cost for TPR-tree.

Varying Grid Level l: The impact of grid-level l on the related pruning benefits of Co-MaxRS-HGrid is illustrated in Figure 3.29b. With increased granularity (i.e., higher l), the amount of pruning also increases. Even for $l = 1$, around 80% $\vec{DO}$ events are pruned in the index level, and 83.25% $\vec{OD}$ objects are pruned during a recomputation. Note that, these values are subsumed by the pruning achieved by our proposed Lemmas in Section 3.2 – the refinement step. But, filtering out significant amount of objects and



(a)

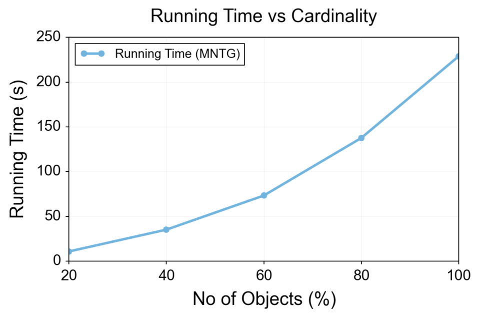


(b)

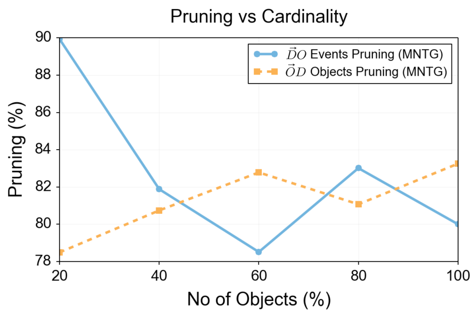
Figure 3.29. Impact of l on (a) Running Time (b) Pruning of Co-MaxRS-HGrid.

events even before accessing the OL data structure improves the processing time – the effect will be more prominent when objects are stored in external storage and/or clouds. In Figure 3.29a, the effect of l on the running-time is shown. Initially, the running-time decreases as l increases (until l=2). The benefits of efficient range queries and pruning obtained by higher level grids overcomes the increased maintenance overhead for $l \leq 2$. Afterwards, the running-time shoots up again for l= 3 and 4 – as the maintenance overhead (i.e., number of grid-change events and cost of partial/full recomputations) for these levels overpowers the minimal benefits gained compared to other lower levels.

Impact of Cardinality: Figure 3.30a presents the effects of cardinality over the running time of Co-MaxRS-HGrid. Processing time is higher for larger datasize as the total number of $\vec{D}O$ and $\vec{O}D$ events increases too. Although, Co-MaxRS-HGrid is always faster compared to Co-MaxRS (without indexing), and slower than Approx-Co-MaxRS for varying datasize (cf. Figure 3.30a and Figure 3.25b). $\vec{D}O$ events pruning slightly reduces, and $\vec{O}D$ objects pruning slightly increases for higher number of objects as illustrated in Figure 3.30b.

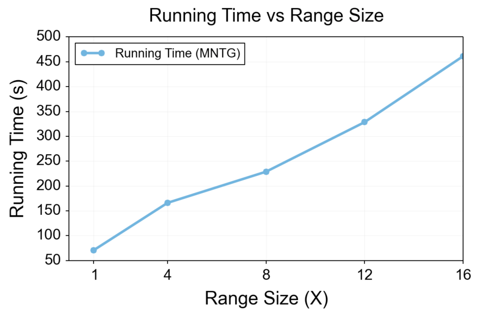


(a)

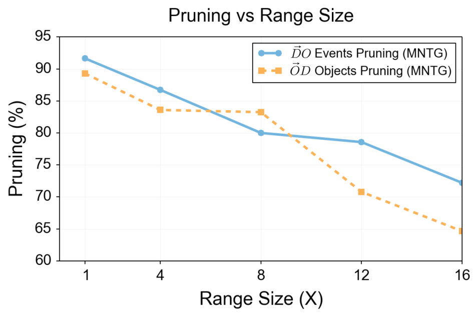


(b)

Figure 3.30. Impact of cardinality on (a) Running Time (b) Pruning of Co-MaxRS-HGrid.



(a)



(b)

Figure 3.31. Impact of range-size on (a) Running Time (b) Pruning of Co-MaxRS-HGrid.

Influence of Range Size: This experiment was designed to observe the impact of range-size on the running time and pruning of Co-MaxRS-HGrid. The effect of pruning for Co-MaxRS-HGrid over different R is similar to the non-indexed Co-MaxRS algorithm (cf. Figure 3.31b and Figure 3.20), i.e., increasing range area (the values on X-axis indicate multiples of the base-size for each dataset) results in fewer portion of events and objects pruned. This also has an adverse effect on the running time of the algorithm. Thus, even

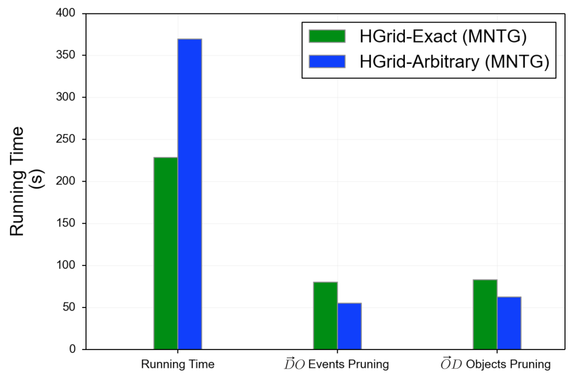


Figure 3.32. Runing time and pruning comparison for using exact R and arbitrayr R.

though increasing range area means fewer grid maintenance overhead, it results in higher processing time for Co-MaxRS-HGrid (see Figure 3.31a).

Performance for Arbitrary R Method: As discussed in Section 3.3.2, if we have a level-l hierarchical grid structure, then the pruning and range query benefits of the index structure is applicable for any R_{arb}, for which $R_l \leq R_{arb} \leq R_0$, where R_0 and R_l represent the size of R for grid-level 0 and l respectively. The idea is to find the lowest numerical level for which the grid size is greater than R_{arb}. This is very useful when building an index on-the-fly based on the given R is not possible. In this experiment, we set R_0 to be $2600 \times 2600m^2$ and R_{arb} to be $2000 \times 2000m^2$. We evaluate the performance (i.e., run-time and pruning) in this setting, compared to the default scheme (i.e., building an index on the fly based on the given R) in Figure 3.32. The running time is 1.6 times more, and around 20-25% fewer objects and events are pruned for HGrid-Arbitrary setting. As expected, performance is poorer than building an R-based index on the fly – although, we note that, the achieved pruning and speed-up in run-time (compared to Co-MaxRS without indexing) is still significant.

3.6. Demonstration: System Design and Demo Details

We implemented a demonstration system, based on the proposed solutions in this chapter to process the Co-MaxRS queries. We now describe the main components of the implemented system architecture and how they interact with each other, and proceed with details of the demo.

Software Architecture: Our final system is a web-based application with interactive and user-friendly interface for both PC and mobile devices, which is implemented using HTML, CSS, and JavaScript (Node.js was used for the server-side programming). We employ *Responsive Web Design* principles in building the website by using CSS media queries, @media rules, and fluid grids—thus, making it suitable to work on every device and screen size. The core Co-MaxRS algorithm is implemented using C++. The software architecture of our system, illustrated in Fig. 3.33, is organized in the following main categories:

• *Query Interface:* The users can select each of the required parameters using the interface by specifying: (1) $\mathbb{F}$ – the query area (depends on the underlying dataset), set dynamically via the zoom-in (+) or out (-) tool; (2) R – specifying d_1 and d_2 values; and (3) T – the time period, specifying t_0 and t_{max}. Users can browse through solutions for each T by sliding the time bar. Additionally, a user will also be able to upload their own dataset, as long as it follows a pre-fixed format – plain file containing tuples of (*object-id, trajectory-id, latitude, longitude, time, weight* (*optional, default=1*)).

• *Visualization Components:* We used Google Maps JavaScript API [**1**] to display a map with respective pins for each object at a particular time instant and the current MaxRS solution. These pins and results will change accordingly as the user drags the time bar.

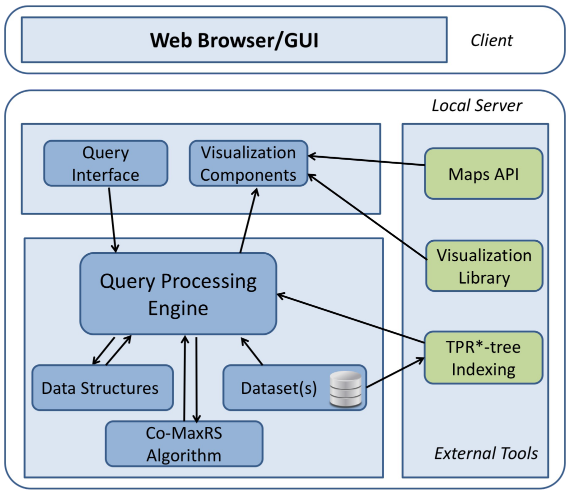


Figure 3.33. Software architecture.

Also, we used various JavaScript visualization tools (such as [**3**]) to enable a different view of the result – trajectories of the Co-MaxRS solutions in 3D spatio-temporal settings.

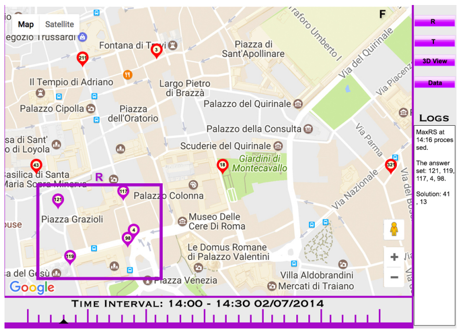
• *Co-MaxRS Algorithm:* Even for small object movements, the optimal location of the query rectangle can change while objects participating in the MaxRS solution stay the same. Instead of maintaining a centroid-location (equivalently, a region) as a Co-MaxRS solution, we maintain a list of objects that are located in the interior of the optimal rectangle placement – the Co-MaxRS answer-set becoming a sequence (*list-of-objects, time-period*). For the example in Fig. 1.1, Co-MaxRS answer-set is: $\{((o_6, o_7, o_8), [t_0, t_1)), ((o_1, o_2, o_3), [t, t_{max})), ((o_1, o_3, o_7, o_8), [t_{max}, t_{max}))\}$. We identified criteria (i.e., critical times) when a particular MaxRS solution is no longer valid, or a new MaxRS solution emerges. The basic algorithm uses KDS (*Kinetic Data Structures*) – maintaining a priority queue of the critical events and their occurrence time, and recomputing MaxRS solutions at each event in order. Recomputing MaxRS is costly, so we devised efficient pruning schemes to: (1) Eliminate the recomputation altogether at certain qualifying critical events; and

(2) Reduce the number of objects that need to be considered when recomputation is unavoidable.

• *Data Structures and Indexing:* We maintain a list for storing each object $o_i \in O$, with its current trajectory, weight, and other necessary information. KDS maintains an event queue, where the events are sorted according to the time-value. Each critical event consists of the related objects, and the occurrence time. Additionally, we also maintained an adjacency matrix to track locality of objects – which is important for smooth processing of our pruning schemes. Moreover, to ensure faster processing over large datasets, we used an existing spatio-temporal data indexing scheme, TPR*-tree (via a C++ library) [**74**].

• *Datasets:* To demonstrate the benefits of our system in different domains, we use several datasets: (1) GPS traces from 500 taxis collected over 30 days in San Francisco Bay area (http://crawdad.org/epfl/mobility/); (2) (*location, time*) information for 370 taxis within Rome for 1 month with sample-period of 7s (http://crawdad.org/roma/taxi/); (3) Human mobility data, in [**88**], where researchers at Microsoft collected data from 182 users in a period of over five years, with 17,621 trajectories; and (4) A small animal movement dataset from (http://crawdad.org/princeton/zebranet/). While the first two datasets are great for demonstrating scalability and traffic-monitoring aspect of our system, the latter two can be used to show applications in human mobility tracing and animal tracking processes. Although all of these datasets had different formats, we converted them into the same format – tuples of (*object-id, trajectory-id, latitude, longitude, time, weight*) values. This enabled our system to handle similarly formatted user-provided data as well.

Demo Specifications: The setup of our demo will consist of a laptop running the web-based application via a web browser. The application is hosted on our server operating

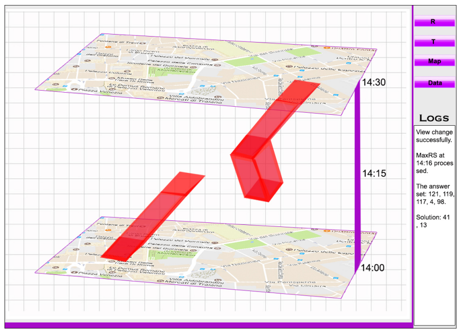


MaxRS at 14:16 processed.

The answer set: 121, 119, 117, 4, 98.

Solution: 41, 13

(a)



View change successfully.

MaxRS at 14:16 processed.

The answer set: 121, 119, 117, 4, 98.

Solution: 41, 13

(b)

Figure 3.34. (a) The main GUI with map and time slider; (b) Answer-set 3D view over T.

at Northwestern University, and will be accessed via a public URL. The demonstration will have three main parts with the following steps:

P1: The first part will have following three main phases:

Phase 1: Specification of the required parameters in the GUI (cf. Fig. 3.34). This phase will show: (a) Selection of appropriate R, $\mathbb{F}$, and T; and (b) Relation between R and $\mathbb{F}$, i.e., how our system dynamically changes R proportionally to $\mathbb{F}$ given an initial value. $\mathbb{F}$ can be dynamically set by the user (+/- tool).

Phase 2: Visualization of the Co-MaxRS result at a certain point of time t using the user-provided parameters over a map (constrained to $\mathbb{F}$). The locations (linearly interpolated at t) of the objects will be displayed via pins on the map, and the solution, i.e., an optimal placement of R and the objects in its interior will be distinctly marked (see Fig. 3.34a).

Phase 3: Visualization of the whole answer-set for the time-period T in spatio-temporal (3D) settings (cf. Fig. 3.34b). Users will be able to analyze how the trajectories of the optimal clusters are evolving over space and time (parallelopipeds).

P2: In the second and most important part of the demo, we will exhibit the benefits of using this tool in analyzing large trajectories data. This part will show: (a) Selection of the area of focus ($\mathbb{F}$) by zooming-in or out; (b) Change the value of t by sliding through per unit time (depends on the data set) within T. We will demonstrate how a user can start from a large $\mathbb{F}$ and relatively larger R, and then continuously refine their region of interest.

P3: The last part of the demo will quickly show steps of **P1** and **P2** for different datasets, emphasizing how the tool can be useful in different domains. Additionally, the steps of uploading a custom dataset will be demonstrated.

CHAPTER 4

Processing Conditional MaxRS in Spatial Data Streams

In this chapter, at first we formulate the problem of C-MaxRS and provide a basic solution (which we use as baseline in later experiments). Subsequently, we extend the basic solution to handle the streaming scenarios. We also devise the changes needed to deal with Weighted C-MaxRS problem. Finally, we propose methods to deal with bursty updates, and present our experimental results.

4.1. Problem Definition

We now introduce the C-MaxRS problem and extend the definition to include the possibility of appearance of new objects, and disappearance of existing ones. In addition, we discuss the concept of submodular monotone functions.

***C*-MaxRS & *C*-MaxRS-DU**: Let us define a set of *POIClass* $K = \{k_1, k_2, \ldots, k_m\}$, where each $k_i \in K$ refers to a class (alternatively, tag and/or type) of the objects, a.k.a. points of interest (POI) . In this setting, each object $o_i \in O$ is represented as a *(location, class)* tuple at any time instant t. We denote a set $X = \{x_1, x_2, \ldots, x_m\}$ as *MinConditionSet*, where $|X| = |K|$ and each $x_i \in \mathbb{Z}+$ denotes the desired lower bound of the count of objects of class k_i in the interior of the query rectangle r – i.e., the optimal region must have at least x_i number of objects of class k_i. Let us assume l_i as the number of objects of type k_i in the interior of r centered at a point p. A utility function $f(O) : \mathcal{P}(O) \to \mathbb{N}_0$, mapping a subset of spatial objects to a non-negative integer is defined as below,

$$f(O) = \begin{cases} (\sum_{i=1}^{|K|} l_i), & \text{if } \forall i \in \{1, 2, 3, ..., |K|\}, l_i >= x_i \\ 0, & \text{if } \exists i \in \{1, 2, 3, ..., |K|\}, l_i < x_i \end{cases}$$

Additionally, we mark O_{r_p} as the set of spatial objects in the interior of rectangle r centered at any point p. Formally, we define:

Definition 3. ***Conditional-MaxRS (C-MaxRS).*** *Given a rectangular spatial field $\mathbb{F}$, a set of objects of interest O (bounded by $\mathbb{F}$), a query rectangle r (of size $a \times b$), a set of POIClass $K = \{k_1, k_2, \ldots, k_m\}$ and a MinConditionSet $X = \{x_1, x_2, \ldots, x_m\}$, the C-MaxRS query returns an optimal location (point) p^* for r such that:*

$$p^* = argmax_{p \in \mathbb{F}} f(O_{r_p})$$

where $O_{r_p} \subseteq O$.

Note that, in the case that there is no placement p for which all the conditions of *MinConditionSet* is met, the query will return an empty answer – indicating to the user to either increase the size of R or decrease the lower bounds for some classes. We now proceed to define C-MaxRS in dynamic scenario – Conditional-MaxRS for Data Updates (C-MaxRS-DU). In a spatial data stream environment, old points of interest may disappear and new ones may appear at any time instant. We can deal with this in two-ways:

- *Time-based:* C-MaxRS is computed on a regular time-interval δ.

• *Event-based:* C-MaxRS is computed on an *event*, where C-MaxRS is maintained (evaluated) every time a new point appears or an old point disappears – both regarded as an *event*.

Although faster algorithms can be developed in time-based settings, the solutions provided would be inherently erroneous for time between t and $t + \delta$. On the other hand, event-based processing ensures that a correct answer-set is maintained all the time. Thus, we deal with the streaming data in event-based manner, for which we denote e^+ as the new point *appearance* and e^- as the old point *disappearance* event. We note that, most of the settings for basic C-MaxRS remains same, except that the set of objects O is altered at each event. We define the set of points of interest in this data stream for any event e as:

$$O_e = \begin{cases} O \cup \{o_e\}, & \text{if } e.type = e^+ \\ O \setminus \{o_e\}, & \text{if } e.type = e^- \end{cases}$$

Formally,

Definition 4. ***Conditional-MaxRS for Data Stream/Updates (C-MaxRS-DU).*** *Given a rectangular spatial field $\mathbb{F}$, a set of objects of interests O (bounded by $\mathbb{F}$), a query rectangle r (of size $a \times b$), a set of POIClass $K = \{k_1, k_2, \ldots, k_m\}$, a MinConditionSet $X = \{x_1, x_2, \ldots, x_m\}$, and a sequence of events E=$\{e_1, e_2, e_3, \ldots\}$ (where each e_i denotes the appearance or disappearance of a point of interest), the C-MaxRS-DU query maintains the optimal location (point) p^* for r such that:*

$$p^* = argmax_{p \in \mathbb{F}} f(O_{r_p})$$

where $O_{r_p} \subseteq O_e$ for every event e in E of the data stream.

Submodular Monotone Function: [**24**] devised solutions to a variant of the MaxRS problem (*best region search*) where the utility function for the given POIs is a submodular monotone function – which is defined as:

Definition 5 (**Submodular Monotone Function**). *If Ω is a finite set, a submodular function is a set function $f : \mathcal{P}(\Omega) \rightarrow \mathbb{R}$ if $\forall_{X,Y} \in \Omega$, with $X \subseteq Y$ and $x \in \Omega \setminus Y$ we have* (1) $f(X \cup \{x\}) - f(X) \geq f(Y \cup \{x\}) - f(Y)$ *and* (2) $f(X) \leq f(Y)$.

In the above definition, (1) represents the condition of submodularity, while (2) presents the condition of monotonicity of the function. In Section 4.2, we will discuss these properties of our introduced utility function $f(O) : \mathcal{P}(O) \rightarrow \mathbb{N}_0$.

Discussion: Note that, for the sake of simplicity, initially we have considered only the counts of POIs when defining the utility function or conditions in X. In Section 4.4, we show that they can be extended to incorporate different non-negative weights for objects – i.e., most of the techniques (including pruning) devised in the work are still applicable with minor modifications. Similarly, although in our provided examples, for brevity, we've only depicted one class per object, the techniques proposed in this work extends to the objects of multiple classes (or tags), e.g., objects can be considered as $(location, classes)$ tuple.

4.2. Basic *C*-MaxRS

In this section, we first convert the C-MaxRS problem to its dual variant and then discuss important properties of the conditional weight function $f(.)$, showing how we can utilize them to devise an efficient solution to process C-MaxRS.

4.2.1. *C*-MaxRS $\rightarrow$ Dual Problem

A naive approach to solve C-MaxRS is to choose each discrete point p iteratively from the rectangular spatial field $\mathbb{F}$ and compute the value of $f(O_{r_p})$ for the set of spatial objects covered by the query rectangle r. As there can be infinite number of points in $\mathbb{F}$, this approach is too costly to be practical. Existing works (see [**58, 24, 36**]) have demonstrated that feasible solutions can be derived for MaxRS (and related problems) by transforming it into its dual problem – *rectangle intersection problem.* A similar conversion is possible for C-MaxRS as well, enabling efficient solutions. In this regards, let R=$\{r_1, r_2, \ldots, r_n\}$ be a set of rectangles of user-defined size $a \times b$. Each rectangle $r_i \in R$ is centered at each point of interest $o_i \in O$, i.e., $|R|$=$|O|$. We define r_i as the *dual rectangle* of o_i. Let us consider a function $g : \mathcal{P}(R) \rightarrow \mathbb{N}_0$ that maps a set of dual rectangles to a non-negative integer. For a set of rectangles $R_k = \{r_1, r_2, \ldots, r_k\}$, let $g(R_k) = f(\{o_1, o_2, \ldots, o_k\})$. Note that, a rectangle is *affected* by a point p if it is in the interior of that rectangle. Let $A(p)$ be the sets of rectangle affected by $p \in \mathbb{F}$. Now, we can redefine C-MaxRS as the following equivalent problem:

Given a rectangular spatial field $\mathbb{F}$*, a set of rectangles* R=$\{r_1, r_2, \ldots, r_n\}$ *(with centers bounded by* $\mathbb{F}$*) where each* r_i *is of a given size* $a \times b$*, a set of POIClass* K=$\{k_1, k_2, \ldots, k_m\}$ *and a MinConditionSet* X=$\{x_1, x_2, \ldots, x_m\}$*, retrieve an optimal location (point)* p^* *such* *that:*

$$p^* = argmax_{p \in P} g(A(p)),$$

where $A(p) \subseteq R$.

The bijection is illustrated with the help of Figure 4.1 using the same example (and conditions) of Figure 1.2. Suppose, rectangles $\{r_1, r_2, r_3, \ldots, r_7\}$ are the dual rectangles of given objects $\{o_1, o_2, o_3, \ldots, o_7\}$ in Figure 4.1, and p_1 and p_2 are two points within the given space. p_1 affects rectangles r_1, r_2, r_3 and p_2 affects r_4, r_5, r_6, r_7, i.e., $A(p_1) = \{r_1, r_2, r_3\}$ and $A(p_2) = \{r_4, r_5, r_6, r_7\}$. Thus, $g(A(p_1))$=$f(\{o_1, o_2, o_3\}) = 3$ as the points conform to the constraints mentioned in Chapter 1, while $g(A(p_2))$=$f(\{o_4, o_5, o_6, o_7\}) = 0$ as they do not.

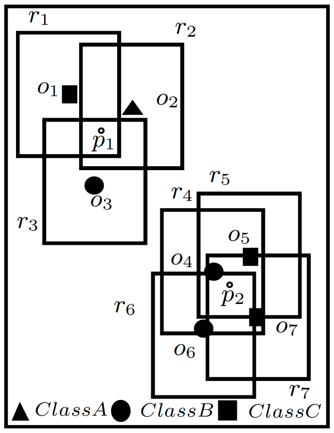


Figure 4.1. C-MaxRS → dual problem.

Similarly, C-MaxRS-DU can be redefined as follows:

Given a rectangular spatial field $\mathbb{F}$*, a set of rectangles* R=$\{r_1, r_2, \ldots, r_n\}$ *(with centers bounded by* $\mathbb{F}$*) where each* r_i *is of a given size* $a \times b$*, a set of POIClass* K=$\{k_1, k_2, \ldots, k_m\}$*, a MinConditionSet* X=$\{x_1, x_2, \ldots, x_m\}$*, and an event* e *(appearance/disappearance of a* *rectangle r_e), update the optimal location (point) p^* such that:*

$$p^* = argmax_{p \in P} g(A(p)),$$

where

$$A(p) \subseteq \begin{cases} R \cup \{r_e\}, & \textit{if } e.type = e^+ \\ R \setminus \{r_e\}, & \textit{if } e.type = e^- \end{cases}$$

4.2.2. Properties of f and g

A method to solve an instance of *Best Region Search* (BRS) problem was devised in [**24**], where the weight function $f : \mathcal{P}(O) \rightarrow \mathbb{R}$ is a submodular monotone function (cf. Definition 5). In [**24**], the problem is first converted to the dual *Submodular Weighted Rectangle Intersection (SIRI)* problem, and then optimization techniques are applied based on these properties of $f(.)$. We now proceed to discuss submodularity and monotonicity of functions $f(O) : \mathcal{P}(O) \rightarrow \mathbb{N}_0$ and $g(R) : \mathcal{P}(R) \rightarrow \mathbb{N}_0$ in our problem settings. We establish two important results for f and g as follows:

Lemma 1. *Both f and g are monotone functions.*

Proof. For a set of spatial objects O,

$$f(O) = \begin{cases} (\sum_{i=1}^{|K|} l_i), & \text{if } \forall i \in \{1, 2, 3, ..., |K|\}, l_i >= x_i \\ 0, & \text{if } \exists i \in \{1, 2, 3, ..., |K|\}, l_i < x_i \end{cases}$$

For any of the classes, if the given lower-bound condition is not met, i.e. $\exists i \in \{1, 2, 3, ..., |K|\}, l_i < x_i$, then $f(O)$=0 for the spatial object set O. However, if all of the conditions are satisfied – i.e., $\forall i \in \{1, 2, 3, ..., |K|\}, l_i < x_i$, then the utility value is equal to the count of spatial objects in O.

Let $O_i \subseteq O_j$. If $O_i = O_j$, $f(O_i) = f(O_j)$, otherwise if $O_i \subset O_j$, there are three possible cases:

Case (a): Both O_i and O_j fail to conform to the *MinConditionSet* X – then $f(O_i) = f(O_j) = 0$.

Case (b): O_j conforms to X, but O_i does not – then $f(O_i) = 0$ and $f(O_j) = |O_j|$. Thus, $f(O_i) < f(O_j)$.

Case (c): Both O_i and O_j conform to X, then $f(O_i) = |O_i|$ and $f(O_j) = |O_j|$. As $O_i \subset O_j$, $|O_i| < |O_j|$, implying, $f(O_i) < f(O_j)$.

We note that there are no possible cases where O_i conforms to X, but O_j does not. Thus, f is a monotone function.

Let R_i and R_j be two sets of dual rectangles generated from the aforementioned two sets of spatial objects – O_i and O_j respectively. Here, $O_i \subseteq O_j \rightarrow R_i \subseteq R_j$. According to the definition of g, $g(R_i) = f(O_i)$ and $g(R_j) = f(O_j)$. As $f(O_i) \leq f(O_j)$, then $g(R_i) \leq g(R_j)$. Thus, g is a monotone function too.

$\square$

Lemma 2. *None of f and g is a submodular function.*

Proof. Let us consider the settings of the preceding proof, i.e., two sets of spatial objects O_i and O_j (where $O_i \subseteq O_j$), and corresponding sets of dual rectangles R_i and

R_j. Suppose, O and R are the set of all objects and dual rectangles respectively. Let us consider a spatial object $o_k \in O \setminus O_j$ and its associated dual rectangle $r_k \in R \setminus R_j$. Then there is a possible case where O_j conforms to X, but neither O_i nor $O_i \cup \{o_k\}$ conform to X. As O_j conforms to X, $O_j \cup \{o_k\}$ will conform too. Thus, $f(O_i) = 0$, $f(O_j) = |O_j|$, $f(O_i \cup \{o_k\}) = 0$, $f(O_j \cup \{o_k\}) = |O_j \cup \{o_k\}| = |O_j| + 1$. Interestingly, we obtain: $f(O_i \cup \{o_k\}) - f(O_i) = 0 - 0 = 0$ and $f(O_j \cup \{o_k\}) - f(O_j) = |O_j| + 1 - |O_j| = 1$; that means $f(O_i \cup \{o_k\}) - f(O_i) < f(O_j \cup \{o_k\}) - f(O_j)$ violating the condition of submodularity. Hence, f is not submodular.
On the other hand, $g(R_i \cup \{r_k\}) - g(R_i) = f(O_i \cup \{o_k\}) - f(O_i) = 0 - 0 = 0$ and $g(R_j \cup \{r_k\}) - g(R_j) = f(O_j \cup \{o_k\}) - f(O_j) = |O_j| + 1 - |O_j| = 1$; which means $g(R_i \cup \{r_k\}) - g(R_i) < g(R_j \cup \{r\}) - g(R_j)$. Thus, g is not submodular too. $\square$

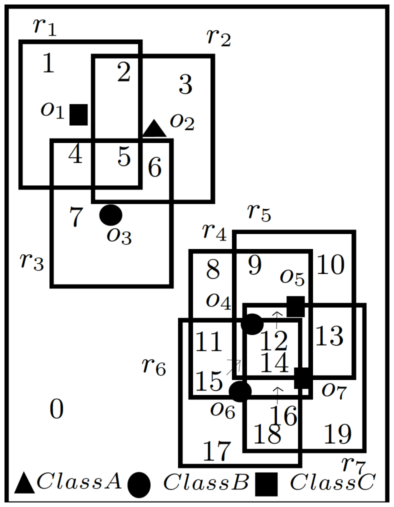
Let us consider the example of Figure 4.1 – suppose O_i=$\{o_4, o_5, o_6, o_7\}$ and two new POIs o_8 and o_9 arrive from class A and C respectively. let O_j=$O_i \cup \{o_8\}$ (i.e., $O_i \subseteq O_j$). Now, $f(O_i)$=(0+2+2)(0)(1)(1)=0 and $f(O_j)$=(1+2+2)(1)(1)(1)=5, i.e., $f(O_i) \leq f(O_j)$, proving monotonicity of f. But $f(O_i \cup \{o_9\})$=(0+3+2)(0)(1)(1)=0 and $f(O_j \cup \{o_9\})$=$(1+3+2)$(1)(1)(1)=6. Thus, $(f(O_i \cup \{o_9\}) - f(O_i) = 0 - 0 = 0) < (f(O_j \cup \{o_9\}) - f(O_j) = 6 - 5 = 1)$, proving non-submodularity of f. Similar examples can be shown for g too.

4.2.3. Processing of C-MaxRS

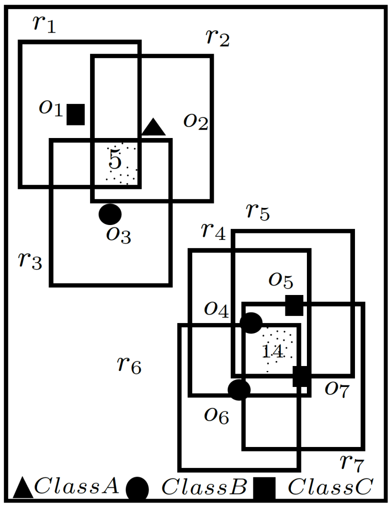
Although f and g are not submodular functions, we show that their monotonicity property can be utilized to derive efficient processing and optimization strategies, similar to the ideas presented in [**24**].

4.2.3.1. Disjoint and Maximal Regions. The edges of the dual rectangles divide the given spatial field into *disjoint* regions where each disjoint region $\mathbb{F}_{d_i}$ is an intersection of a set of rectangles. Consider the examples shown in Figure 4.2(i). Rectangles $\{r1, r2, ..., r7\}$ divided the space into distinct regions numbered $0-19$, e.g., region 0 is the region outside all rectangles, and region 14 is the intersection of rectangles $\{r4, r5, r6, r7\}$. Intuitively, all points in a single disjoint region $\mathbb{F}_{d_i}$ affects the same set of rectangles, i.e., $A(p)$ is same for all $p \in \mathbb{F}_{d_i}$. There could be at most $O(n^2)$ disjoint regions (shown in [**24**]). To compute C-MaxRS, a straightforward approach can be to iterate over all the $O(n^2)$ disjoint regions (one point from each region) and choose the optimal one – thus reducing the search space into a finite point set. For example, we only need to evaluate 20 points for the settings of Figure 4.2(i).

A disjoint region $\mathbb{F}_{d_i}$ is termed as a *maximal region* $\mathbb{F}_{m_i}$ if: (1) it is rectangular, and (2) its left, right, bottom, top edges are (respectively) the parts of the left, right, bottom and top edges of some dual rectangles of R. In Figure 4.2(ii), region 5 and 14 are maximal regions. For example, the left, right, bottom, and top edges of region 5 is a part of the corresponding edges r_2, r_1, r_1, r_3 respectively. [**24**] showed that for each distinct region $\mathbb{F}_{d_i}$, there exists a maximal region $\mathbb{F}_{m_i}$ such that $A(\mathbb{F}_{d_i}) \subseteq A(\mathbb{F}_{m_i})$. Using this idea, and the fact that $g(.)$ is monotonic, we can shrink the possible search space to only the set of all maximal regions. As an example (see Figure 4.2), region 4 and 5 are affected by $R_1 = \{r_1, r_3\}$ and $R_2 = \{r_1, r_2, r_3\}$ respectively. As $R_1 \subset R_2$, so by the monotonicity of g, $g(R_1) \leq g(R_2)$. So, only evaluating $g(R_2)$ is sufficient instead of evaluating both $g(R_1)$ and $g(R_2)$. Though there could still be $O(n^2)$ maximal regions in the worst case, the actual number in practice is much lower (compared to disjoint regions).



(i)



(ii)

Figure 4.2. (i) Disjoint & (ii) Maximal regions

4.2.3.2. Maximal Slabs and Slices. A *maximal slab* is the area between two horizontal lines in the space where the top line passes along the top edge of a dual rectangle and bottom one passes along the bottom edge of a dual rectangle, and the area between two horizontal lines contains no top or bottom edge of any other dual rectangles. In Figure 4.3(i), there are three maximal slabs, enclosed by the top and bottom edges of rectangles $\{r_3, r_1\}$, $\{r_4, r_3\}$ and $\{r_6, r_5\}$ (top edges are solid line, and bottom edges are dotted lines). According to [**24**], each maximal region intersects at least one maximal slab – i.e., the solution space can be reduced to the interior of all the maximal slabs only. As maximal slabs are defined based on one top and one bottom edge of dual rectangles, there could be at most $O(n)$ maximal slabs.

All the maximal slabs can be retrieved using a horizontal sweep line algorithm in a bottom-up manner. A set is maintained to keep track of the rectangles intersecting the current slab, and a *flag* to indicate the type of the last horizontal edge processed. When the sweep line is at the bottom (top) edge of a rectangle, it is inserted into (deleted from) the set and *flag* is set to bottom (top). Additionally, when processing a top edge of a rectangle, the algorithm checks whether a maximal slab is encountered (i.e., currently *flag*=bottom). We can compute the upper bound for a slab by applying $g(.)$ on the rectangles intersecting that slab, i.e., if R_{s_i} is the set of rectangles that intersects slab $\mathbb{F}_{s_i}$, then the upper bound of $g(p)$ for any point $p \in \mathbb{F}_{s_i}$ is $g(R_{s_i})$. For example, in Figure 4.3(i), $\{r_4, r_5, r_6, r_7\}$ intersect the bottommost slab. So, the upper bound for that slab is $g(\{r_4, r_5, r_6, r_7\}) = 0$ (as no members of class A present – not conforming to the introduced constraints in Chapter 1).

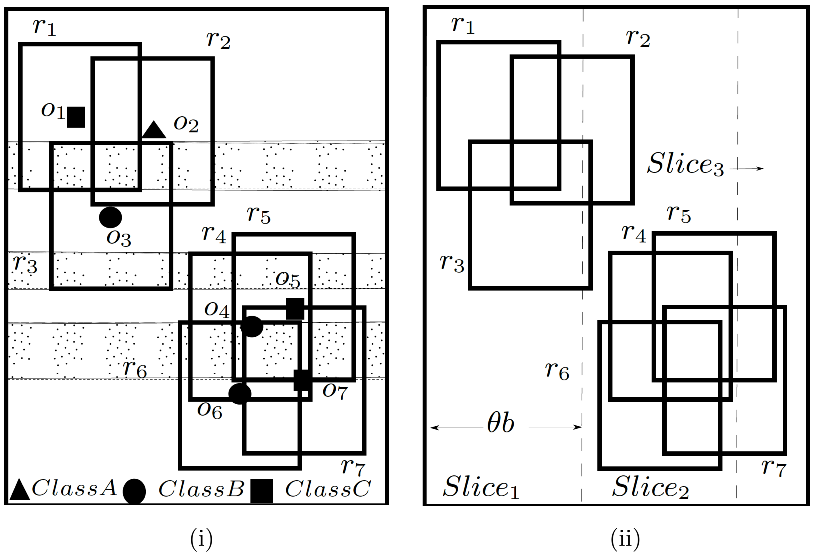


Figure 4.3. (i) Maximal Slabs & (ii) Slices

Finally, the monotonicity of g allows us to adapt another optimization technique introduced in [**24**] – *slices* (see Figure 4.3(ii)). The idea is to divide the whole space into vertical slices (along x-axis). The width of the slices is query-dependent, i.e., $\theta \times b$, where θ is a real positive constant value ($\theta > 1$ and optimal value can be tuned empirically) and b is the width of the query rectangle r. After dividing the space into slices, we retrieve the slabs within each slice using the horizontal sweep-line algorithm described above and obtain upper-bound of a slice by computing the maximum upper-bound among all the slabs within that slice. We can then process the slices in a greedy manner – sort them in order of their upper-bounds and process one by one until the currently obtained result is greater than the upper-bounds of the remaining slices. Similar greedy approach can be adopted to process the maximal slabs within each slice. As an example, suppose there are four slices $\{s_1, s_2, s_3, s_4\}$ with upper bounds $\{8, 3, 5, 2\}$ respectively. The order in which the slices will be processed is: $\{s_1, s_3, s_2, s_4\}$. Assume that after processing s_1, current optimal g value is 3. So there is a possibility the optimal solution within s_3 might exceed the current overall optimal solution of 3. After processing s_3, if the result is 4, then processing s_2 and s_4 is unnecessary. Slices allow more pruning than slabs, and also still $O(n)$ maximal slabs is processed in all the slices (see [**24**]).

4.3. C-MaxRS in Data Streams

Given an efficient solution based on the dual variant of the problem and the properties of the utility function, we now proceed with introducing novel techniques to deal with more realistic scenarios, i.e., data arriving in streams with the possibility of objects appearing and disappearing at different time instants. Using the approach of the basic C-MaxRS problem presented in previous section as a foundation, we augment the solution with compact data-structures and pruning strategies that enable effective handling of data streams environment.

4.3.1. Data Structures

Before proceeding with the details of the algorithms and pruning schemes, we describe the data structures used. We introduce two necessary data structures: quadtree (denoted $QTree$) and a self-balanced binary search tree (denoted $SliceUpperBoundBST$), and describe the details of our representation of slices. We re-iterate that while [**24**] tackled the problem of best-placement with respect to an aggregate function, we are considering different constraints – class membership. In addition, we do not confine to a limited time-window. This is why, in addition to the quadtree used in [**24**], we needed self-balancing binary tree to be invoked as dictated by the dynamics of the modifications.

4.3.1.1. $QTree$**.** We need to process a large number of (variants of) range queries when computing f for any point, i.e., finding intersecting rectangles for a given rectangle. To ensure this is processed efficiently, we use quadtree ([**66**]) – a tree-based structure ensuring fast ($O(\log n)$) insertion, deletion, retrieval and aggregate operations in 2D space. $QTree$ recursively partitions $\mathbb{F}$ into four equal sized rectangular regions until each leaf only contains one POI. The $QTree$ for our running example settings is shown in Figure 4.4.

4.3.1.2. $SliceUpperBoundBST$**.** Recall that the algorithm proposed in Section 4.2.3 iterates through the slices in decreasing order of their maximum possible utility values (upper-bounds). To achieve this for basic C-MaxRS, sorting the slices in order is sufficient ($O(n \log n)$ operation). However, given the possibility of appearance (e^+) and disappearance (e^-) events in dynamic streaming scenarios, the upper-bounds of slices (and their respective order) may change frequently with time. To deal with these efficiently, we introduce a balanced binary search tree ($SliceUpperBoundBST$, see [**60**]) in

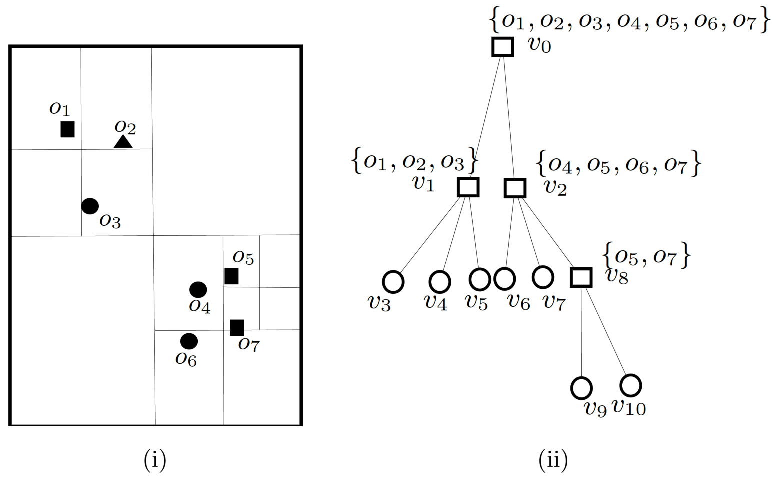
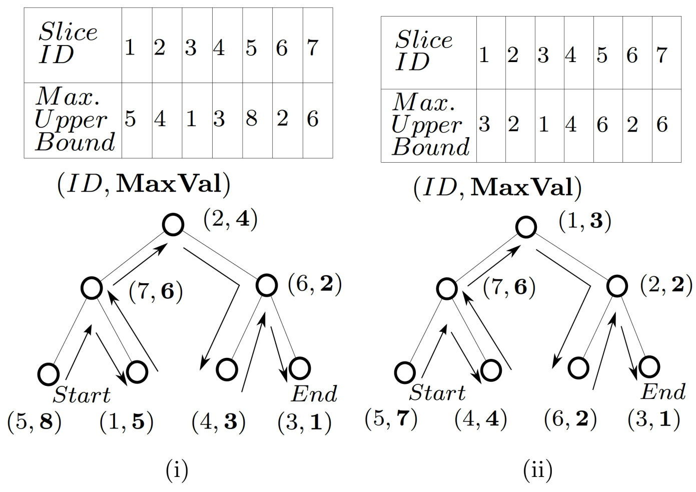


Figure 4.4. (i) Quadtree division & (ii) *QTree*

our data structures instead of maintaining a sorted list whenever an event occurs. Different kinds of self-balancing binary search tree (e.g., AVL tree, Red-black tree, Splay tree, etc.) can be used for this purpose. We used AVL tree in our implementation. If there are ϵ number of dynamic events and n number of slices, sorting them on each event would incur a total of $O((\epsilon + 1)n \log n)$ time-complexity. Whereas we can build a balanced BST *SliceUpperBoundBST* initially in $O(n \log n)$, and update the tree at each event in $O(\log n)$ time. Thus the total cost of maintaining the sorted slices via *SliceUpperBoundBST* is $O(n \log n + \epsilon \log n)$ time. As in real-world applications running for a long time, we would incur large values of both ϵ and n, in which case, using *SliceUpperBoundBST* is much more efficient.

To traverse the slices in decreasing order via *SliceUpperBoundBST*, an in-order traversal from left to right order is needed (assuming, higher values are stored on the left children), and vice versa. *SliceUpperBoundBST* arranges the slices based on their upper bounds of g. In Figure 4.5, a sample slice structure (of 7 slices) and their respective maximum utility upper bounds (dummy values) are shown for two events at different times



Slice ID	1	2	3	4	5	6	7
Max. Upper Bound	5	4	1	3	8	2	6

Slice ID	1	2	3	4	5	6	7
Max. Upper Bound	3	2	1	4	6	2	6

Figure 4.5. *SliceUpperBoundBST* at time (i) t_1 & (ii) t_2

t_1 and t_2. The corresponding *SliceUpperBoundBST* structure for both cases is shown as well. The process of accessing the slices in decreasing order (an in-order traversal) is demonstrated in Figure 4.5(ii).

4.3.1.3. List of Slices. We use a list S_{slice} to maintain the slices and their related information. Each slice $s_i \in S_{slice}$ is represented as a 6−tuple $(id, R, S_{slabs}, p_c, lazy, maxregsearched)$. These fields are described as follows:

• *id*: A numeric identification number for the slice.

• R: The set of rectangles currently intersecting with the corresponding slice.

• S_{slabs}: The set of maximal slabs in the interior of the slice.

• p_c: The local optimum point within the slice.

• *lazy*: This field is used to reduce computational overhead in certain scenarios. While processing streaming data, there are cases when an e^+ or e^- event may alter the local solution (optimal point) for a particular slice, but overall, the global solution is guaranteed to remain unchanged. In those cases, we will not re-evaluate the local processing of that slice (i.e., pruning) – rather will set the *lazy* field to *true*. Later, when the possibility of a global solution change arises – local optimal points are re-processed for all the *lazy* marked slices to sync with the up-to-date state. Initially, *lazy* fields for all slices are set to $false$.

• *maxregsearched*: This field is used to indicate whether the slice's local solution is up-to-date or not. *maxregsearched* is set to *true* when the corresponding slice is evaluated and its local maximal point is stored in p_c. Initially, *maxregsearched* is set to $false$ for all the slices. While processing C-MaxRS by iterating through the slices, all the slices with this field set to *true* are not re-evaluated (skipped).

4.3.2. Base Method

In this section, we start by introducing two related functions (sub-methods), and then proceed with describing the details of the base method to process C-MaxRS using the ideas discussed so far.

4.3.2.1. PrepareSlices(S_{slice}). Function 7 takes S_{slice} as input and sets up different fields of each slice accordingly. For each slice $s_i \in S_{slice}$, their respective R and S_{slabs} are computed (lines 2-3), and other variables are properly initialized (lines 4-6). In line 3, the maximum upper bounds of g (denoted g_{maxub}) among all the slices is retrieved as well, while *ScanSlab* is the horizontal sweep-line procedure discussed in Section 4.2.3.2. *SliceUpperBoundBST* is also build via line 7.

Time-Complexity. : While analyzing time-complexities, we will denote number of slices $|S_{slice}|$ as s and number of rectangles (and objects too) as n. Suppose all of the slices in S_{slice} is passed to Function 7 for processing. In worst case scenario, line 2 takes $O(n)$ time. [**24**] shows *Scanslab*() (i.e., line 3) aggregately takes at most $O(n)$ time for all the slices together. Any *SliceUpperBoundBST* operations (cf., line 4) need $O(\log s)$ time. Thus, the overall time-complexity of Function 7 is $O(s(n + \log s) + n)$ – or, $O(sn)$ (as typically, $n > s$).

Algorithm 7 PrepareSlices(S_{slice})

```
Require: A set of slices $S_{slice}$
1: for each $s_i$ in $S_{slice}$ do
2:     $s_i.R \leftarrow$ the set of rectangles currently intersecting with $s.i$
3:     $(s_i.S_{slabs}, g_{maxub}) \leftarrow ScanSlab(s_i.R)$
4:     $SliceUpperBoundBST.update(s_i.id, g_{maxub})$
5:     $s_i.p_c \leftarrow null$
6:     $s_i.lazy \leftarrow false$
7:     $s_i.maxregsearched \leftarrow false$
8: end for
```

4.3.2.2. SliceSearchMR(p_c^*). Function 8 takes the current global maximal point p_c^* as input and returns the updated solution. The function iterates through all the slices via in-order traversal of *SliceUpperBoundBST* from the *root* (lines 1-2). The process is terminated if g_{maxub} of the current slice is $\leq$ of current maximum utility value $g(A(p_c^*))$ (lines 3-4), or when all the slices are evaluated. At each iteration, we check whether there exists an already computed solution (unchanged) for the slice. If so, we avoid recomputing it (lines 6-7), otherwise we retrieve the current optimal solution for the slice and update related variables accordingly (lines 9-11). Finally, we update the global optimal point by comparing it with the local solution (lines 12-13).

Time-Complexity. : In the worst case scenario, all the nodes in *SliceUpperBoundBST* are traversed in Function 8. A stack based implementation of in-order traversal takes $O(s)$ time, and computing the $g()$ function can take up to $O(n)$ time. Thus, the overall worst-case complexity for Function 8 is $O(sn)$.

Algorithm 8 SliceSearchMR(p_c^*)

```
Require: Global maximal point $p_c^*$
Ensure: Updated global maximal point $p_c^*$
 1: $c_{node} \leftarrow SliceUpperBoundBST.root$
 2: while inorder traversal of $SliceUpperBoundBST$ from $c_{node}$ is not done do
 3:   if $c_{node}.g_{maxub} \leq g(A(p_c^*))$ then
 4:     break
 5:   else
 6:     if $S_{slice}[c_{node}.slice_id].maxregsearched = true$ then
 7:       $p_{local}^* \leftarrow S_{slice}[c_{node}.slice_id].p_c$
 8:     else
 9:       $p_{local}^* \leftarrow$ Compute local optimal point
10:       $S_{slice}[c_{node}.slice_id].p_c \leftarrow p_{local}^*$
11:       $S_{slice}[c_{node}.slice_id].maxregsearched \leftarrow true$
12:     end if
13:     if $g(A(p_{local}^*)) > g(A(p_c^*))$ then
14:       $p_c^* \leftarrow p_{local}^*$
15:     end if
16:   end if
17: end while
18: return $p_c^*$
```

4.3.2.3. SolveCMaxRS. Algorithm 9 presents the base method $SolveCMaxRS$ that retrieves the optimal point p_c^* from a snapshot of the database. p_c^*, $QTree$ and $SliceUpperBoundBST$ are initialized, and the dual rectangles of the given POIs O is computed in lines 1-4. In lines 5-6, we update the $QTree$ by inserting all the dual rectangles in the structure. Line 7 retrieves the list of slices using the given width θb. Finally, the method uses Function 7 to initialize the fields of slices properly in line 8, and computes the C-MaxRS solution using Function 8 in line 9.

Time-Complexity. : Initializing and inserting all the rectangles in the quadtree takes $O(n \log n)$ time along with a random initialization of $SliceUpperBoundBST$ in $O(s)$. Listing all the slices (line 7) also takes $O(s)$ time. Using the complexities of $PrepareSlices()$

and $SliceSearchMR()$ from previous discussion, we can conclude that worst-case time complexity of Algorithm 9 is $O(n \log n + sn)$.

Algorithm 9 SolveCMaxRS(O, a, b)

```
Require: A set of objects $O$, query size $a \times b$
Ensure: An optimal point $p_c^*$
 1: $p_c^* \leftarrow null$
 2: $QTree.init()$
 3: $SliceUpperBoundBST.init()$
 4: $R \leftarrow$ the set of $a \times b$ rectangles centered at each $o \in O$
 5: for each $r \in R$ do
 6:     $QTree.insert($new $Noder)$
 7: end for
 8: $S_{slice} \leftarrow$ list of slices of width $\theta b$
 9: $PrepareSlices(S_{slice})$
10: $p_c^* \leftarrow$ SliceSearchMR($p_c^*$)
11: return $p_c^*$
```

4.3.3. Event-based Pruning

Recall that, to cope with the challenges of real-time dynamic updates of the point space via data streams, we opted for the event-driven approach rather than the time-driven approach. Our goal is to maintain correct solution by performing instant updates during an event. In case of spatial data streams, a straightforward approach is to use Algorithm 9 whenever an event occurs. We now proceed to identify specific properties/states of events (both e^+ and e^+) that allow us to prune unnecessary computations while processing them. Note that, in this settings, a bunch of e^+ and e^- events can occur at the same time.

4.3.3.1. Pruning in e^-. To derive an optimization technique for e^- events, let us first establish few related important results.

Lemma 3. *Removal of a rectangle r_e (object o_e) from the point space $\mathbb{F}$ never increases the value of $g(A(p))$ (correspondingly $f(A(p))$), $\forall p \in P$.*

Proof. Denote the removed rectangle as r_e. We consider two cases: • $r_e \in A(p)$: After the removal of r_e, the set of rectangles affected by p becomes $A(p) \setminus \{r_e\}$. Now, $A(p) \setminus \{r_e\} \subset A(p)$. Hence, from Theorem 1, $g(A(p) \setminus \{r_e\}) \leq g(A(p))$. Thus, the removal in this case does not increase $g(A(p))$.
• $r_e \notin A(p)$): After removal of r_e, the set of rectangles affected by p is still $A(p)$. Hence, $g(A(p))$ remains unchanged. In this case as well, the removal does not increase $g(A(p))$. Similarly, we can show a proof for removing an object – i.e., o_e from $\mathbb{F}$. □

Lemma 3 paves the way for the pruning of slices from being considered a solution at e^- events.

Lemma 4. *The maximum utility point (global solution) p_c^* is unchanged after the removal of a rectangle r_e from the space $\mathbb{F}$ if $r_e \notin A(p_c^*)$.*

Proof. Here, $r_e \notin A(p_c^*)$. Suppose, after removing r_e, $A'(p_c^*)$ rectangles are affected by p_c^*. Note that, $A'(p_c^*)$=$A(p_c^*)$ (as $r_e \notin A(p_c^*)$), implying $g(A'(p_c^*)) = g(A(p_c^*))$. Thus, the utility values of p_c^* remains the same. By Lemma 3, the removal of r_e does not increase the utility value of $p, \forall p \in P$. Suppose, the utility value of a point p, ($p \in P$ and $p \neq p_c$), are $g(A(p))$ and $g'(A(p))$ respectively before and after the removal of r_e, then $g'(A(p)) \leq g(A(p))$. Again, p_c^* being the maximal point, $g(A(p)) \leq g(A(p_c^*))$, $\forall p \in P, p \neq p_c^*$. Above mentioned inequalities imply that $g'(A(p)) \leq g(A'(p_c^*))$, $\forall p \in P, p \neq p_c^*$, meaning p_c^* remains unchanged. □

Using Lemma 4, we can prune local slice processing at an e^- event, if $r_e \notin A(p_c^*)$, i.e., we need to only update $QTree$ in this case.

Lemma 5. *The utility value of the maximal point p_c^* is changed after the removal of a rectangle r_e if $r_e \in A(p_c^*)$.*

Proof. If p_c^* is returned as the maximal point, then $g(A(p_c^*)) > 0$ (i.e., we have a solution). After the removal of r_e, the set of rectangles affected by p_c^* becomes $A(p_c^*)-\{r_e\}$. There are two possible cases:

• $A(p_c^*) - \{r_e\}$ conforms to X: In this scenario, $g(A(p_c^*)) - g(A(p_c^*) - \{r_e\}) = |A(p_c^*)| - (|A(p_c^*)| - 1) = 1$.

• $A(p_c^*) - \{r_e\}$ does not conform to X: Here, $g(A(p_c^*)) - g(A(p_c^*)) - \{r_e\}) = |A(p_c^*)| - 0 = |A(p_c^*)|$

In both cases, $g(A(p_c^*))$ is changed. □

Lemma 5 implies that, if a rectangle removed at an e^- event is in $A(p_c^*)$, we need to re-evaluate local solutions for the respective slice(s), and update global maximal point if necessary.

Lemma 6. *Suppose a point space P is divided into a set of slices S_{slice}, and the slice containing the maximum utility point p_c^* is s_{max}. Let, S_s be another set of slices, where $S_s \subset S_{slice}$ and $s_{max} \notin S_s$. Subsequently, the removal of a rectangle r_e spanning through only the slices in S_s, i.e., affecting only the local maximum utility values of s_i, $\forall s_i \in S_s$, does not have any effect on the global maximum utility point p_c^*.*

Proof. Let p_{local}^* be the maximum utility point of a slice $s_i \in S_s$. $\forall p \in s_i$ where $s_i \in S_s$, $g(A(p_c^*)) \geq g(A(p_{local}^*))$ and $g(A(p_{local}^*)) \geq g(A(p))$. According to Lemma 3, after the removal of r_e, for any $s_i \in S_s$, $g(A(p - \{r_e\})) \leq g(A(p))$. From the above three inequalities, we can deduce: $\forall p \in s_i$ where $s_i \in S_s$, $g(A(p) - \{r_e\}) \leq g(A(p_c^*))$. This holds true $\forall s_i \in S_s$. Thus, p_c^* still remains the maximum utility point (as s_{max} is not altered), and s_{max} is still the slice containing p_c^*. □

Lemma 6 implies that, if the slice containing global maximal point p_c^* is unchanged while some other slices are altered, then following the update of $QTree$, we can delay the processing of altered slices at that time instance as it is not going to affect the global maximal answer anyway. For this reason, we incorporated the *lazy* field in each slice. In this case, we set *lazy* to *true* for each of these altered slices, indicating that they should be re-evaluated later only when the slice containing global maximal point is altered.

4.3.3.2. Pruning in e^+. During an e^+ event, a rectangle (object) appears in the given space $\mathbb{F}$. We now present two lemmas, based on which we derive pruning strategies at e^+ events.

Lemma 7. *Addition of a rectangle r_e (object o_e) in the given space $\mathbb{F}$ never decreases the value of $g(A(p))$ (correspondingly $f(A(p))$), $\forall p \in P$.*

Proof. Let the added rectangle be r_e. We consider two cases: • $r_e \in A(p)$: After the addition of r_e, the set of rectangles affected by p becomes $A(p) \cup \{r_e\}$. Now, $A(p) \subset A(p) \cup \{r_e\}$. Hence, from Theorem 1, $g(A(p) \cup \{r_e\}) \geq g(A(p))$. So, in this case $g(A(p))$ does not decrease.

• $r_e \notin A(p))$: After addition of r_e, the set of rectangles affected by p still remains $A(p)$. Hence, $g(A(p))$ does not change as well. Thus, $g(A(p))$ does not decrease in this scenario as well.

Similarly, we can show a proof for adding an object – i.e., o_e to $\mathbb{F}$. $\square$

For e^- events, we leveraged on ideas like Lemma 3 – i.e., removal of a rectangle never increases utility value of a point, to devise clever pruning schemes depending on the fact that local or global maximal points are guaranteed to be unchanged in certain scenarios. But, for e^+ events, those are not applicable as addition of a rectangle *may* increase utility of affected points. Interestingly, though, there are scenarios when the utility values are unchanged, e.g., when $A(p)$ does not conform to X. Also, as shown in the 2nd case of the proof of Lemma 7 – we only process a slice if its affected by the addition of r_e.

Lemma 8. *Suppose, we have a set of classes $K = \{k_1, k_2, \dots, k_m\}$, and are given corresponding MinConditionSet $X = \{x_1, x_2, \dots, x_m\}$. Let R be the set of rectangles overlapping with a slice $s_i \in S_{slice}$, and let l_i be the count of rectangles of class k_i in R. Then, addition of a rectangle r_e of class k_i has no effect on the local maximal solution of s_i if:*
(1) $x_i - l_i \geq 2$, or
(2) $(\exists l_j \neq l_i)\ x_j - l_j \geq 1$

Proof. (1) In this settings, the maximum possible utility value of s_i before addition of r_e is 0. Because, even if for a point $p \in s_i$, $A(p) = R$, then $g(A(p))$=0 as $l_i < x_i$ and R does not conform to X. After the addition of r_e, suppose the count of class k_i objects in R is l'_i, i.e., l'_i=$l_i + 1$. As given $x_i - l_i \geq 2$, then $l'_i < x_i$. Thus, R still does not conform to X, and maximum possible utility value of s_i remains 0.
(2) Similarly, the maximum possible utility value of s_i before addition of r_e is 0. Because,

even if for a point $p \in s_i$, $A(p) = R$, then $g(A(p))$=0 as $l_j < x_i$ for $\exists l_j \neq l_i$, and R does not conform to X. After the addition of r_e of class k_i, l_j remains unchanged. Thus, R still does not conform to X, and maximum possible utility value of s_i remains 0. □

Lemma 8 lays out the process of pruning during an e^+ event. For each slice, we maintain an integer value *diff* (i.e., $x_i - l_i$) per class in K denoting whether the corresponding upper-bound for that class has been met or not. When adding a rectangle of class k_i, for each affected slices, we first check whether $\mathit{diff}_i \geq 2$, and if so – we just update diff_i and skip processing that slice. Similarly, if $\mathit{diff}_i \leq 1$, but for $\exists \mathit{diff}_j \geq 1$, we can skip the slice. For example, suppose we have a setting of three classes A, B, C where X=$\{2, 3, 5\}$. Suppose a slice contains $\{2, 1, 4\}$ members of respective classes. In this case, arrival of a rectangle of class B or C has no effect on that slice. We incorporate these ideas in our Algorithm 11 (although, for brevity, we skip details of implementing and maintaining *diff* in algorithms).

4.3.4. Algorithmic Details

We now proceed to augment the ideas from the previous section in our base solution. In this regard, we provide the details of two algorithms $SolveCMaxRS^-$ and $SolveCMaxRS^+$, implementing the ideas of pruning in e^- and e^+ events respectively.

4.3.4.1. $SolveCMaxRS^-$**.** In Algorithm 10, we present the detailed method for maintaining C-MaxRS result during an e^- event using the ideas introduced in Section 4.3.3.1. Firstly, r_e is retrieved (from o_e) and then deleted from then $QTree$ is updated accordingly (cf. lines 1-2). Subsequently, in lines 3-4, all the slices intersecting with r_e is retrieved and the set of slices marked lazy (S_{lazy}) is initialized. Lines 5-8 iterate through all the

Algorithm 10 SolveCMaxRS$^-$ $(e^-(o_e), a, b, p_c^*)$

```
Require: An $e^-(o_e)$ event, query size $a \times b$, and current maximal point $p_c^*$
Ensure: Updated maximal point $p_c^*$
 1: $r_e \leftarrow$ the $a \times b$ rectangle centered at $o_e$
 2: $QTree.delete(r_e)$
 3: $S_e \leftarrow$ set of slices intersecting $r_e$
 4: $S_{lazy} \leftarrow$ set of slices marked lazy
 5: for each $s_i \in S_e$ do
 6:     if before the removal $r_e \in A(s_i.p_c^*)$ then
 7:         $s_i.lazy \leftarrow true$
 8:         $S_{lazy} \leftarrow S_{lazy} \cup \{s_i\}$;
 9:     end if
10: end for
11: $s_{max} \leftarrow$slice containg global $p_c^*$
12: if before the removal $r_e \in A(s_i.p_c^*)$ then
13:     $PrepareSlices(S_{lazy})$
14:     $p_c^* \leftarrow$ SliceSearchMR$(p_c^*)$
15: end if
16: return $p_c^*$
```

affected slices one by one and check for each of them to see if the local maximal point $s_i.p_c^*$ is affected by r_e – if so, it marks them as lazy for future update and also adds them to S_{lazy}. If the slice containing global maximal point i(i.e., s_{max}) is not affected, then the processing of slices in S_{lazy}i skipped (pruning) in lines 9-12. Otherwise, if pruning is not possible, necessary computations are carried out in lines 11-12.

Time-Complexity. : Deleting from a quadtree takes $O(\log n)$ time (line 2). Listing all the intersecting and lazy slices in worst cases will generate $O(s)$ computations (lines 3-4). Iterating over all the overlapped slices and computing $g()$ takes up $O(sn)$ times in worst case (lines 5 - 8). If pruning is not possible, the complexities of $PrepareSlices()$ and $SliceSearchMR()$ adds up too (lines 10-12). The overall worst-case time complexity of Algorithm 10 is $O(sn + s + \log n + sn + sn)$ – or, in short, $O(sn)$.

4.3.4.2. ***SolveCMaxRS$^+$*.** In Algorithm 11, we initially retrieve the dual rectangle r_e associated with the event and update $QTree$ by inserting r_e as a new node in lines 1-2. Then, the set of slices affected by r_e is computed and S_{lazy} is initialized in lines 3-4. We introduce a Boolean variable $isPrunable$ in line 5 to track whether Lemma 8 can be applied or not. Lines 6-10 iterate through all the affected slices one by one, an checks: if $s_i.R$ now conforms to X and makes change accordingly (modifies $isPrunable$), and sets up $s_i.lazy$ and list S_{lazy} properly. Lines 11-12 prunes the event if conditions of Lemma 8 is satisfied, i.e., if $isPrunable = true$ then the global maximal p_c^* needs no update. Otherwise, it processes C-MaxRS on the snapshot (lines 13-14).

Time-Complexity. : The analysis of lines 1-4 here is similar to Algorithm 10. Iterating over all the intersecting slices and checking the constraints takes up $O(s \times |X|)$ times in worst case. So, if pruning is possible, the time-complexity of Algorithm 11 is $O(s \times |X| + s + \log n)$ time (faster than pre-pruning stage of Algorithm 10). But, in worst case, if pruning is not possible, then the complexity will be $O(sn)$ (similar to Algorithm 10).

4.4. Weighted C-MaxRS

In the discussions so far, we only considered the counting variant of the C-MaxRS problem, i.e., the weights of each participating object are all equal to 1 (or, any other fixed value). While we have noted the portability of the results, in this section, we explicitly show how the algorithms and pruning schemes proposed thus far should be modified to cater to the case when the objects can have different weights. Firstly, we appropriately revise the definitions of f, g, and C-MaxRS-DU to allow different weights, and show that it does not affect the monotonicity and non-submodularity of f and g. Subsequently,

Algorithm 11 SolveCMaxRS$^-$ $(e^-(o_e), a, b, p_c^*)$

```
Require: An $e^+(o_e)$ event, query size $a \times b$, and current maximal point $p_c^*$
Ensure: Updated maximal point $p_c^*$
 1: $r_e \leftarrow$ the $a \times b$ rectangle centered at $o_e$
 2: $QTree.insert$(new $Node r_e$)
 3: $S_e \leftarrow$ set of slices intersecting $r_e$
 4: $S_{lazy} \leftarrow$ set of slices marked lazy
 5: $isPrunable \leftarrow true$
 6: for each $s_i \in S_e$ do
 7:    if after the addition $R \cup r_e$ conforms to $X$ then
 8:       $s_i.lazy \leftarrow true$
 9:       $isPrunable \leftarrow false$
10:       $S_{lazy} \leftarrow S_{lazy} \cup \{s_i\}$
11:    end if
12: end for
13: if $isPrunable = true$ then
14:    return $p_c^*$
15: end if
16: $PrepareSlices(S_{lazy})$
17: $p_c^* \leftarrow$ SliceSearchMR($p_c^*$)
18: return $p_c^*$
```

we outline the modifications for the pruning schemes for the weighted version. While there are no major changes incurred in the fundamental algorithmic aspects, we note that weights may have impact on the pruning effects, as illustrated in Section 4.6.

4.4.1. Redefining f, g, and C-MaxRS-DU

$\boldsymbol{f^w}$: Let us define a set of *POIClass* $K = \{k_1, k_2, \ldots, k_m\}$, where each $k_i \in K$ refers to a class of objects. Suppose, $O = \{o_1, o_2, \ldots, o_n\}$ is the set of objects (POIs), and the set $W = \{w_1, w_2, \ldots, w_n\}$, where $w_i > 0, \forall w_i \in W$, contains the weight values of all POIs, i.e., the weight of an object o_i is w_i. In this setting, each object $o_i \in O$ is represented as a *(location, class, w_i)* tuple at any time instant t. We denote a set $X = \{x_1, x_2, \ldots, x_m\}$

as *MinConditionSet*, where $|X| = |K|$ and each $x_i \in \mathbb{R}+$ denotes the desired lower bound of the weighted-sum of the objects of class k_i in the interior of the query rectangle r, i.e., $\sum_{o_i \in r \wedge o_i.class=k_i}^{\forall o_i} w_i$. Thus, the optimal region must have objects of class k_i whose weights add up to at least x_i. Let us define l_i^w, a non-negative real number, for a given set of objects O as follows:

$$l_i^w = \sum_{o_j \in O \wedge o_j.class=k_i}^{\forall o_j} w_j$$

.

Subsequently, we can define a utility function $f^w(O) : \mathcal{P}(O) \to \mathbb{N}_0$, mapping a subset of spatial objects to a non-negative integer is defined as below,

$$f^w(O) = \begin{cases} (\sum_{i=1}^{|K|} l_i^w), & \text{if } \forall i \in \{1, 2, 3, ..., |K|\}, l_i^w >= x_i \\ 0, & \text{if } \exists i \in \{1, 2, 3, ..., |K|\}, l_i^w < x_i \end{cases}$$

.

C-MaxRS-DU: Let us denote the rectangle r centered at point p as r_p, and O_{r_p} as the set of spatial objects in the interior of r_p. We can now define C-MaxRS-DU as follows (including the weights):

Definition 6. ***Conditional-MaxRS for Data Updates (C-MaxRS-DU).*** *Given a rectangular spatial field $\mathbb{F}$, a set of objects of interests O (bounded by $\mathbb{F}$) and their corresponding set of weight values W, a query rectangle r (of size $a \times b$), a set of POIClass $K = \{k_1, k_2, \ldots, k_m\}$, a MinConditionSet $X = \{x_1, x_2, \ldots, x_m\}$, and a sequence of events $E = \{e_1, e_2, e_3, \ldots\}$ (where each e_i denotes the appearance or disappearance of a point of interest), the C-MaxRS-DU query maintains the optimal location (point) p^* for*

r such that:

$$p^* = argmax_{p \in \mathbb{F}} f^w(O_{r_p})$$

where $O_{r_p} \subseteq O_e$ for every event e in E of the data stream.

$\boldsymbol{g^w}$: Similar to the function g, we can introduce g^w as a bijection of f^w, i.e., for a set of rectangles $R_k = \{r_1, r_2, \ldots, r_k\}$, let $g^w(R_k) = f^w(\{o_1, o_2, \ldots, o_k\})$. $g^w : \mathcal{P}(R) \rightarrow \mathbb{R}_0$ maps a set of dual rectangles to a non-negative real number (weighted-sum).

4.4.2. Monotonicity and non-submodularity of f^w and g^w

As we define $w_i \in W$ as a positive real number, the weighted-sum of a set of objects – $\sum_{o_i} w_i$, is also a positive real number. This is similar to the counting variant of the problem. Thus, using the similar logic as Lemma 1 and Lemma 2, we derive the following:

Lemma 9. *Both f^w and g^w are monotone functions.*

Lemma 10. *None of f^w and g^w is a submodular function.*

The proofs follow the similar intuition as the corresponding proofs of Lemma 1 and Lemma 2 and are omitted –however, we proceed with discussing their implication in a more detailed manner next.

4.4.3. Discussion

Lemma 9 and Lemma 10 show that the properties of the utlity functions remain same, for both counting and weighted version. Subsequently, we can derive the following:

Lemma 11. *Removal of a rectangle r_e (object o_e) from the point space $\mathbb{F}$ never increases the value of $g^w(A(p))$ (correspondingly $f^w(A(p))$), $\forall p \in P$.*

Lemma 11 can be proved in similar way as the proof of Lemma 3, as f^w and g^w are also monotonous. Thus, Lemma 11 validates the other necessary lemmas (i.e., Lemma 4, 5, and 6) related to the e^- pruning scheme. This shows that we can solve the problem of an e^- event, for an object o_e (rectangle r_e) and its weight w_e, by using the same algorithm $SolveCMaxRS^-$. For the e^+ event, we present the following lemmas: (skipping proof for brevity)

Lemma 12. *Addition of a rectangle r_e (object o_e) in the given space $\mathbb{F}$ never decreases the value of $g^w(A(p))$ (correspondingly $f^w(A(p))$), $\forall p \in P$.*

Lemma 13. *Suppose, we have a set of classes $K = \{k_1, k_2, \ldots, k_m\}$, and are given corresponding $MinConditionSet$ $X = \{x_1, x_2, \ldots, x_m\}$. Let R be the set of rectangles overlapping with a slice $s_i \in S_{slice}$, and let l_i^w be the weighted-sum of rectangles of class k_i in R. Then, addition of a rectangle r_e of class k_i has no effect on the local maximal solution of s_i if:*
(1) $x_i - l_i^w > w_e$, or
(2) $(\exists l_j \neq l_i)$ $x_j - l_j^w > 0$

Lemmas 12 and 13 demonstrates that an e^+ event, for an object o_e (rectangle r_e) and its weight w_e, can be processed similarly via $SolveCMaxRS^+$ algorithm.

4.5. C-MaxRS in Bursty Updates

In many spatial applications, the data streaming rate often varies wildly depending on various external factors – e.g., the time of the day, the need of the users, etc. A peculiar phenomenon in such cases is the, so called, bursty streams – which is, the streaming rate becomes unusually high and a large number of objects appearing or disappearing in a very short interval. In such scenarios, instead of processing every single update, we assume that the update streams are gathered for a period of time. The C-MaxRS-DU algorithm is based on sequential processing of events, and thus, its efficiency is particularly sensitive to the bursty input scenario. In this section, we first briefly discuss the challenges of processing bulk of events via Algorithm 10 and 11, and argue that a different technique is necessary to deal with the new bursty events in an aggregated manner. Subsequently, we propose additional data-structures and a new algorithm, C-MaxRS-Bursty, to maintain C-MaxRS during bursty streaming updates scenarios. Finally, we briefly discuss how our proposed scheme can be utilized in a distributed manner, for the purpose of further improvements in scalability.

4.5.1. Challenges

As per the algorithms presented in Section 4.3, Algorithm 10 ($SolveCMaxRS^{-}$) and Algorithm 11 ($SolveCMaxRS^{+}$) are used to deal with any new e^{-} and e^{+} event, respectively. The worst case time complexity of both the algorithms is $O(sn)$. Let us denote γ as the average streaming (a.k.a. bulk-updates) rate during a bursty stream scenario, i.e., γ events (both e^{+} and e^{-}) occur simultaneously per time instance. In this setting, the worst-case complexity of processing these events using C-MaxRS-DU is $O(\gamma sn)$. We note that, due to the effectiveness of the pruning schemes, the average processing time is considerably faster than the worst case complexity presented here (details in Section 4.6). However, the overhead of performing Algorithm 10 and Algorithm 11 γ times is still significant, specially when fast and accurate responses are required. For example, line 3 of Algorithm 10 takes $O(s)$ time to find the slices S_e that intersect with the new event rectangle r_e. Instead of computing this γ times (i.e., $\gamma \times O(s)$), it would be better if we scan the list of slices only once, and retrieve all the slices that are affected by the new γ events in one pass. Moreover, if the slice containing global p_c^*, i.e., s_{max}, is affected by multiple events, then $PrepareSlices()$ and $SliceSearchMR()$ would be redundantly processed multiple times. Hence, the intuition is that we can get rid of these overheads by dealing with the bursty events aggregately.

To this end, we propose an additional data structure (e.g., a spatial index) and devise an efficient algorithmic solution. In Section 4.6, we demonstrate via experimental observations that, for a sufficiently large value of γ, C-MaxRS-Bursty outperforms the event-based processing scheme by an order of magnitude. The basic idea is as follows: we first create a modified slice-based index, S_{index} for newly occurring γ events (appearing or disappearing objects). Then, we directly add/remove these new events over the existing slice structure S_{slice} in one iteration, and check the pruning conditions for each slice only once. We describe these ideas in the following section.

4.5.2. Additional Data Structures

The first step, when handling bursty data updates, is to index the new events based on the locations of their related objects. This allows us later to efficiently retrieve all the new events related to each slice $s_i \in S_{slice}$. In this regard, any well-known indexing scheme may be used, e.g., R-tree, Quad-tree, Grid indexing [**61, 72**], etc. To take advantage of the already introduced slices data structure, we propose to use slice-based indexing for the new data. Slice indexing is, basically, a special version of the $p \times q$ grid-indexing – where $q = 1$. Suppose, S_{index} represents the slice index of new appearing/disappearing objects. Then, we can create S_{index} as a duplicate of S_{slice}, i.e., width of each slice in S_{index} is also $\theta \times b$ (where, $\theta > 1$) and $|S_{index}| = |S_{slice}| = s$. An example of the proposed slice indexing is given in Figure 4.6. Suppose, there are 10 new events occurring at the same time – 7 e^+ and 3 e^-, and there are three slices which enclose these event locations. Note that, by event location, we mean the location of the appearing/disappearing object related to the event. In Figure 4.6, $Slice_1, Slice_2, Slice_3$ has respectively 3, 4, 3 new events falling within their boundary.

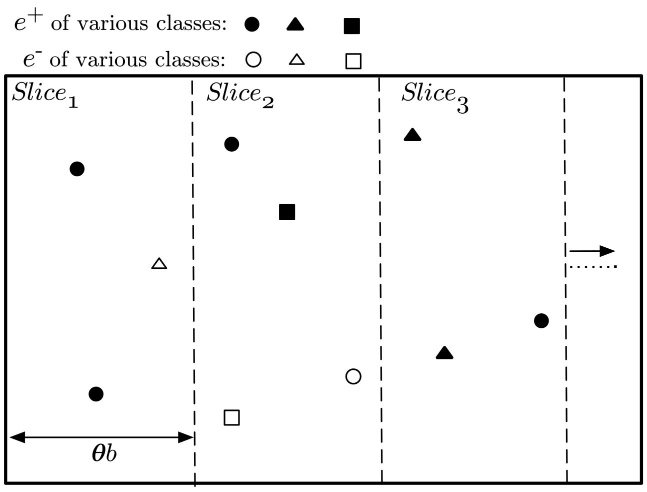

e^+ of various classes: ● ▲ ■
e^- of various classes: ○ △ □
$Slice_1$
$Slice_2$
$Slice_3$
θb

Figure 4.6. Slice indexing over new data.

As described in Section 4.3, each of the slices in S_{slice} track the corresponding rectangles intersecting with them, in addition to the list of maximal slabs, local optimum points and the other attributes. S_{index}, in turn, indexes new events over the slices. An event e, corresponding an object o_e, is exclusively enclosed by exactly one slice in S_{index}, although the rectangle r_e can overlap with multiple slices. This is illustrated in Figure 4.7. Based on this, we can divide the interior of each slice into three regions (cf. Figure 4.7):

• **Left-overlapping Region (*lr*):** Rectangles of events in this region overlaps with the left neighboring slice. Width of lr is $\frac{b}{2}$. In Figure 4.7, events in lr of $Slice_2$ impact the processing of $Slice_1$ too.

• **Non-overlapping Region (*nr*):** Rectangles of events in this region are fully enclosed within the slice itself. nr is $(\theta - 1) \times b$ wide, i.e., always non-empty as $\theta > 1$.

• **Right-overlapping Region (*rr*):** Rectangles of events in this region overlaps with the right neighboring slice. Width of rr, similar to lr, is $\frac{b}{2}$. In Figure 4.7, events in rr of $Slice_2$ is also a part of the processing of $Slice_3$.

Based on the discussion above, each slice $s_i \in S_{index}$ is represented as a $4-$tuple $(seq\_num, E_{lr}, E_{nr}, E_{rr})$. The role of each attribute is as follows:

• seq_num: An integer value assigned to the slice. This encodes the boundary of the slice. For a slice s_i, the horizontal extent of s_i is represented by $[(seq\_num_i - 1) \times \theta b, seq\_num_i \times \theta b)$.

• E_{lr}: The set of new events in the lr region of the slice.

• E_{nr}: The set of new events in the nr region of the slice.

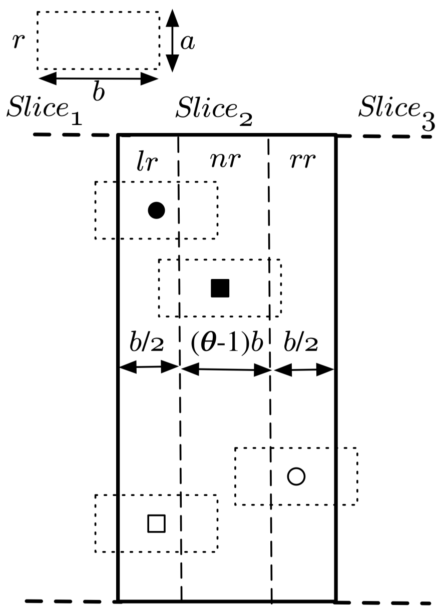


Figure 4.7. Regions within a slice.

• E_{rr}: The set of new events in the rr region of the slice.

Note that, both S_{index} and S_{slice} can be merged into one giant slice data structure during implementation. We present them as separate structures here, so that the background motivation and complexity analysis can be clearly demonstrated in the text, i.e., the objective of these two structures are different – S_{slice} divides the space and overall computation in small slices, while S_{index} is used only to efficiently index a set of new events.

4.5.3. Processing Bursty Updates

When a collection of new e^+ and e^- events occur at a time instant, the first step is to initialize and built the slice index S_{index}. Function 12 shows the steps used to build the index from scratch over the new data. In line 1, S_{index} and *seq_num* of its slices are initialized. The other attributes of each slice $s_i \in S_{index}$ is initialized in lines 2 – 3, i.e., all

Algorithm 12 BuildIndex(E_{new}, θ, b)

Require: A set of new events E_{new}, slice-width constant θ, and query width b
Ensure: Newly build index S_{index}

```
 1: $S_{index} \leftarrow$ initialize a list of slices and their $seq\_num$ (width $= \theta b$)
 2: for each $s_i$ in $S_{index}$ do
 3:     $s_i.E_{lr}, s_i.E_{nr}, s_i.E_{rr} \leftarrow \{\ \}$
 4: end for
 5: for each $e$ in $E_{new}$ do
 6:     $s_e \leftarrow$ the slice $o_e$ is in
 7:     if $o_e \in s_e.lr$ then
 8:         $s_e.E_{lr} \leftarrow s_e.E_{lr} \cup e$
 9:     else if $o_e \in s_e.rr$ then
10:         $s_e.E_{rr} \leftarrow s_e.E_{rr} \cup e$
11:     else
12:         $s_e.E_{nr} \leftarrow s_e.E_{nr} \cup e$
13:     end if
14: end for
15: return $S_{index}$
```

event lists (based on the region) are set to an empty list. Lines 4 – 11 iterate though each new events from E_{new} and set the index attributes accordingly. In line 5, the function retrieves the slice to which o_e belongs, which can be computed in $O(1)$ time. Lines 6 – 11 find which region o_e is in, and add the corresponding event to the appropriate list. Finally, the newly created index S_{index} is returned in line 12. The operations from lines 1 – 3 takes $O(s)$ time, and lines 4 – 11 takes $O(\gamma)$ time, where γ is the buursty updates rate. The processing cost of Function 12 is $O(\gamma) + O(s)$. If we assume $\gamma > s$, then the overall time-complexity is $O(\gamma)$.

Algorithm 13 shows the steps of our approach for handling a set of new bursty events E_{new}, where $|E_{new}| = \gamma$. We combine the pruning ideas of Algorithm 10 and 11, and ensure that *PrepareSlices* and *SliceSearchMR* functions are only called once for these γ new events. In line 1, we use the *BuildIndex* function to prepare the slice index over the new data in $O(\gamma)$ time. We initialize S_{lazy}, $isPrunable$, and $prev$ in lines 2 – 4. The idea is to traverse the slices from S_{slice} in one direction, e.g., from left to right. The main idea is that for each slice s_i of S_{slice}, we retrieve the required information of new events from the slice-index S_{index}. The goal is to make sure that we query information of each slice from S_{index} only once throughout the process. In this regard, we maintain 3 variables – $prev$, cur, and $next$ – representing the $seq\_num - 1$, seq_num, and $seq\_num + 1$ slices from S_{index} (new information) any time. Initially, in lines 4-6, cur is set to the left-most slice, and $prev$ is set to null as there is no slice before that value of cur.

Lines 7 - 29 iterate though each of the slices s_i from S_{slice} in order (e.g., left to right). At first, information for the $(i + 1)$-th slice index is retrieved into $next$. In line 9, all the related new events of s_i is stored in $E_{cur_{slice}}$, which is the union of new objects in cur region, and $prev.rr$ and $next.lr$ region (cf. Figure 4.7). In line 10, we check if there are any new events that overlap with the current slice s_i – otherwise we move on to the next slice. In lines 12 – 15, we iterate through the e^+ events of s_i — retrieve r_e, insert r_e in the $QTree$ and add r_e to $s_i.R$ for each of them. Similarly, lines 16 – 23 iterate over the e^- events of s_i, although r_e is deleted from $QTree$ and $s_i.R$ in this case. Also, lines 20 – 22 ensure that $s_i.lazy$ is set to $true$ and s_i is added to S_{lazy} if r_e overlaps with the local optimum solution. Lines 28 and 29 updates the $prev$ and cur variables appropriately, and line 30 retrieves the slice s_{max} containing the global solution. We need to recompute global solution whenever $s_{max}.lazy = true$ or $isPrunable = false$ (cf. lines 31 - 33). Finally, the newly computed (or, if pruned, the old) p_c^* is returned in line 34. In Algorithm 13, each new event is only processed at most 2 times, because $\theta > 1$ and a rectangle r_e can only overlap with at most two slices. Thus, the overall time-complexity of lines 1- 30 of

Algorithm 13 is $O(\gamma)$. Also, $PrepareSlices$ and $SliceSearchMR$ is only processed once for all the new events, instead of worst case γ times via Algorithms 10 and 11. For large values of N, the overall processing time of Algorithm 13 is consumed by the execution time of $PrepareSlices$ and $SliceSearchMR$.

4.5.4. Discussion

We presented a slice-based simplified indexing scheme in this section to process a set of bursty events. As slice-indexes are a specialized grid-indexing [**61**], they can be implemented both as main-memory or external-memory based. We implemented the proposed slice-indexing in main memory for our experiments. The reason is two- fold. (1) Many recent works have shown that main-memory indexes are usually necessary to provide high update and build performance [**72**] – which is paramount in dealing with bursty updates scenarios; and (2) In our experiments, we vary γ from 100 to 100k — which can be stored in-memory. Although, we note that, in extreme scenarios (e.g., Facebook users) where the no of total objects as well as bursty objects surpass the main memory storage capacity of servers, external memory implementations and parallel processing of indexes would be necessary. Many works such as [**42, 45**] presented parallel processing techniques for R-trees and range queries. [**42**] developed a simple hardware architecture consisting of one processor with several disks to parallelize R-tree processing, where R-tree code is identical to the one for a single-disk R-tree with minimal modifications. [**90**] proposed a novel architecture named VegaGiStore, to enable efficient spatial query processing over big spatial data and concurrent access, via distributed indexing and map-reduce [**19**] technique. Recently, SpatialHadoop [**22**] provides a library to perform map-reduce based parallel processing for many spatial operations, including R-tree and grid indexing. We can modify the grid indexing parameters for SpatialHadoop to convert it into a slice-indexing in a straightforward manner. We note that, Hadoop [**71**] and map reduce procedure has a significant overhead, i.e., SpatialHadoop slice indexing will only be useful if there are a huge number of bursty events, as well as a lot of resources (Hadoop nodes) available.

4.6. Experimental Study

In this section, we evaluate the performance of our algorithms. To show the effectiveness of our approach we compare it with the baseline. Since there are no existing solutions, to evaluate our solutions to the C-MaxRS-DU problem, we extended the best known MaxRS solution to cater to C-MaxRS-DU (see Section 4.3 – i.e., processing the C-MaxRS at each event without any pruning) and used it as a baseline. For bursty streams, we compare the performance of C-MaxRS-Bursty and C-MaxRS-DU, i.e., C-MaxRS-DU becomes the baseline then.

Dataset: Due to user privacy concerns and data sharing restrictions, very few (if any) authentic large categorical streaming data (with accurate time information) is publicly available. Thus, we used synthetic datasets in our experiments to simulate spatial data streams. Data points are generated by using both Uniform and Gaussian distributions in a two-dimensional data space of size $1000m \times 1000m = \mathit{1km}^2$. To simulate the behavior of spatial data streams from these static data points, we use exponential distribution with mean inter-arrival time of $10s$ and mean service time of $10s$. Initially, we assume that 60% of all data points have already arrived in the system, and use this dataset for static part of evaluation. The remaining 40% of the data points arrive in the system by following exponential distribution as stated earlier. Any data point that is currently in the system, can depart after being served by the system. For experiments related to C-MaxRS-Bursty, we select γ no of events (either in Gaussian or uniform distribution) at any time instant to emulate bursty inputs.

Parameters: The list of parameters with their ranges, default values and symbols are shown in Table 4.1.

Parameter Name & Symbol	Possible Values	Default Value
Object distribution	Uniform, Gaussian	Gaussian
No. of objects, N	10k, 20k, 30k, 40k, 50k, 60k, 70k, 80k, 90k, 100k, 200k	50k
No. of POIClass, β	3, 4, 5, 6, 7	5
Min count (per class), μ	1, 2, 3, 4, 5	3
Query area, λ (in m^2)	100, 225, 400, 625, 900	400
Theta, (θ)	1, 2, 3, 4, 5	3
Shape of R, $b : a$	0.25,0.5,1,2,4	1
Weight, w_i	[1, 10]	1
Bursty updates rate, γ	100, 250, 500, 1k, 2.5k, 5k, 10k, 20k, 30k, 40k, 50k, 60k, 70k, 80k, 90k, 100k	1k

Table 4.1. Parameters

Settings: We have used Python 3.5 programming language to implement our algorithms. All the experiments were conducted in a PC equipped with intel core i5 6500 processor and 16 GB of RAM. We measure the average processing time of monitoring C-MaxRS in various settings. We also compute the performance of Static C-MaxRS computation. In the default settings, the processing time for Static C-MaxRS is 85.86 s. Note that, we exclude the processing time for static C-MaxRS computation in further analysis as this part is similar for both baseline and our approach.

The datasets and the code used in the experiments are publicly available at: http://www.cs.northwestern.edu/∼mmh683/project-works/cMaxRS-ds.html.

4.6.1. Performance Evaluation: Event-based Scenario

We now present our detailed observations over different combinations of the parameters for non-bursty scenario (i.e., C-MaxRS-DU).

4.6.1.1. Varying Number of Objects, N. In this set of experiments, we vary number of objects, N, from 10K to 100K, and compare our algorithm with the baseline for different N using both Gaussian and Uniform distributions. Figure 4.8(i) shows that for Gaussian distribution, the average processing time for our approach (in seconds) increases quadratically (semi-linearly) with the number of objects, whereas the processing time of baseline increases exponentially with the increase of N. For Gaussian distribution, on average our approach runs 3.08 times faster than the baseline algorithm. For Uniform distribution, on an average our approach runs 3.23 times faster than the baseline algorithm (Figure 4.8(ii)). We also observe that our approach outperforms the baseline in a greater margin for a large number of objects as processing time of our approach increases linearly with N for Uniform distribution.

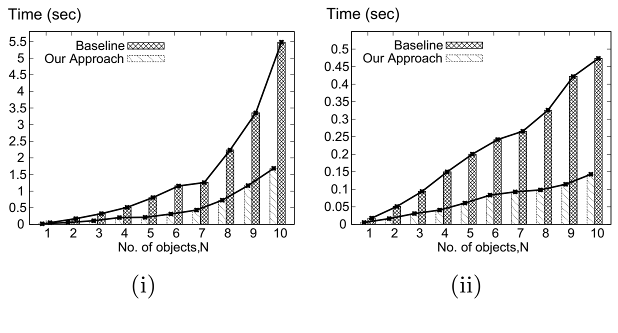
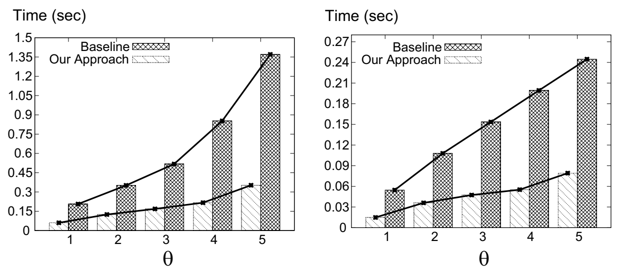


Figure 4.8. Varying N (i) Gaussian (ii) Uniform.

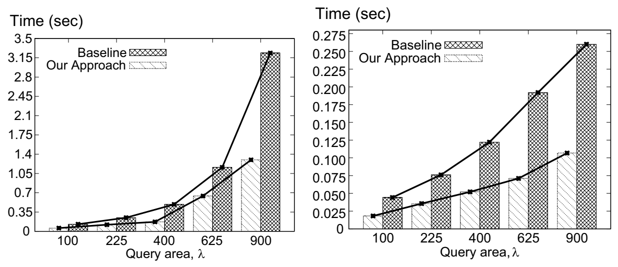
4.6.1.2. Varying Theta (θ). Figure 4.9 compares the performance of our approach with the baseline by varying theta (θ) for Gaussian and Uniform distributions. We observe that for both distributions the processing time of baseline algorithm increases at a higher rate



(i) (ii)

Figure 4.9. Varying θ (i) Gaussian (ii) Uniform.

than our algorithm, with the increase of θ. Moreover, in all the cases, our approach significantly outperforms the baseline algorithm in the absolute scale/sense. On the average, our approach runs 3.37 and 3.31 times faster than the baseline in Gaussian and Uniform distributions, respectively.



(i) (ii)

Figure 4.10. Varying λ (i) Gaussian (ii) Uniform.

4.6.1.3. Varying λ - the Area of the Query Rectangle. The impact of varying the area of the query rectangle on the average processing times of our approach and baseline algorithm, is shown in Figure 4.10(i) and Figure 4.10(ii). For Gaussian distribution, on an average our approach shows 2.22 times better performance than the baseline approach. Similarly, in Uniform distribution, our approach runs 2.25 times (on average) faster than the baseline. Additionally, note that, as the area of query rectangle increases, corresponding processing time increases as well – due to the possibility of a dual rectangle intersecting with more slices (and other dual rectangles).

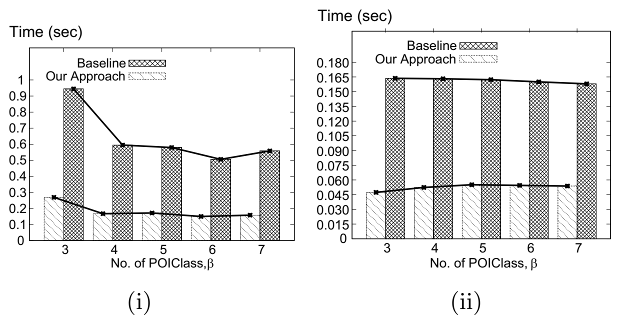


Figure 4.11. Varying β (i) Gaussian (ii) Uniform.

4.6.1.4. Varying POIClass Count, β. The average processing time of our approach and the baseline for varying POIClass Count, β is shown in Figure 4.11 (Gaussian (i) and Uniform (ii)). We observe that the processing time is maximum for the initial case where POIClass Count, β is minimum. Also, we can see that for the both distributions, the processing time decreases with increasing value of β – i.e., handling larger number of classes is faster. On an average our approach runs 3.45 times faster than the baseline algorithm for Gaussian distribution of dataset. In case of Uniform distribution of data, our approach runs 3.06 times faster than the baseline.

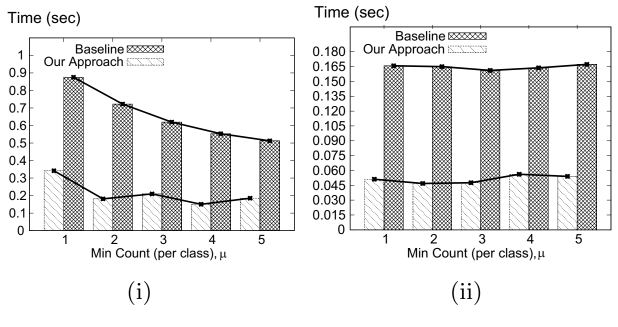
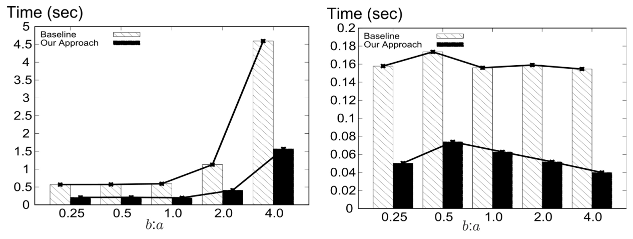


Figure 4.12. Varying μ (i) Gaussian (ii) Uniform.

4.6.1.5. Varying Min Class Count, μ. Figure 4.12 shows the average processing time of our approach and the baseline by varying Min Class Count, μ. Figures show that for both Gaussian and Uniform distributions, our approach outperforms the baseline significantly. We observe that on an average our approach runs 3.09 and 3.21 times faster than the baseline for Gaussian and Uniform distributions of dataset, respectively. We also note that, the processing time for our approach is largely unaffected by the varying μ values.



(i) (ii)

Figure 4.13. Varying $b : a$ (i) Gaussian (ii) Uniform.

4.6.1.6. Varying Shape of R, $b : a$. By default, we have used $b : a = 1$ in other experiments, i.e., R is square-shaped. In this experiment, we investigate whether varying the shape of R, i.e., changing the ratio between its width and height, has any effects on the processing time of C-MaxRS-DU. In Figure 4.13(i) for Gaussian distribution, as width (b) of R is increased, the processing time increases too. This is because, we use $\theta \times b$ as the slice width and as b increases, number of slices s decreases – reducing the benefits of spatial subdivision. Interestingly, similar trend is not observed in the uniform settings. We note that, in all cases, our approach runs faster than the baseline. In case of Uniform distribution our approach outruns the baseline approach by 2.99 times on average. In case of Gaussian distribution, our approach outruns the baseline approach by 2.82 times on average.

4.6.1.7. Comparing Pruning Rules. In this set of experiments, we compare the performance of the different components of our approach. First, we have extended the static C-MaxRS algorithm to handle spatial data streams, which we call the baseline. Then we introduce two pruning rules, one for the appearance event, e^+-Pruning and the other for disappearance event, e^--Pruning. Finally, we combine both pruning rules to design our approach.

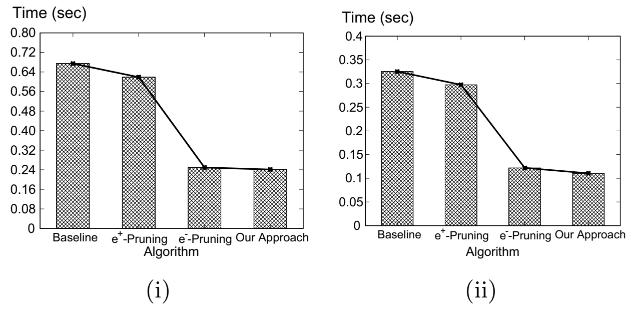


Figure 4.14. Comparing pruning rules (Unweighted Objects)(i) Gaussian (ii) Uniform.

From the figure we can see that e^+-Pruning scheme gives 8.25% performance gain from the baseline algorithm for Gaussian distribution and gives 8.56% performance gain from the baseline algorithm for Uniform distribution of data. The e^--Pruning scheme provides almost 62.49% performance gain from the baseline for Uniform distribution and 63.01% performance gain from the baseline algorithm for Gaussian distribution.

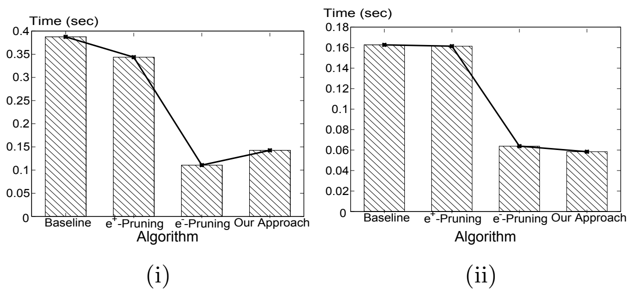


Figure 4.15. Comparing pruning rules (Weighted Objects) (i) Gaussian (ii) Uniform.

We also perform this experiment using weighted objects, where each object is assigned with a random weight. We vary the weights of the objects from 1 to 10. In Figure 4.15, we see similar trends among the evaluated algorithms. Also, we note that, the processing time is faster for the weighted experiments. It is because, due to the variance in the weights of objects, more events can be pruned easily. This experiment also validates our analysis in Section 4.4.

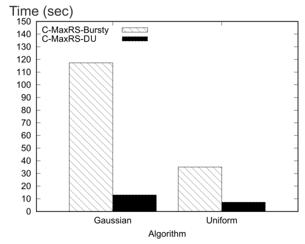


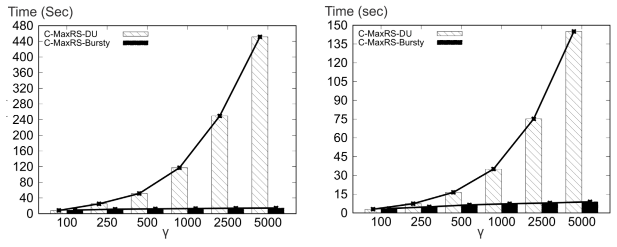
Figure 4.16. Comparing C-MaxRS-DU and C-MaxRS-Bursty for default settings.

4.6.2. Performance Evaluation: Bursty Streaming Updates

We now present our detailed observations over different combinations of the parameters for bursty updates (i.e., C-MaxRS-Bursty vs C-MaxRS-DU).

4.6.2.1. C-MaxRS-Bursty vs C-MaxRS-DU. We present the performance comparison (over both distribution of data) for C-MaxRS-DU and C-MaxRS-Bursty in Figure 4.16 in default settings (i.e., $\gamma = 1000$). We can see that C-MaxRS-Bursty is way more efficient than C-MaxRS-DU in handling bursty streams in both distributions, i.e., C-MaxRS-Bursty is almost 5 and 10 times faster than C-MaxRS-DU in the default settings for uniform and Gaussian distribution of data, respectively.

4.6.2.2. Varying γ. We change the value of the bursty streaming rate, γ, from 100 to 5000. Figure 4.17 shows the total processing time (in seconds) of γ events together. In

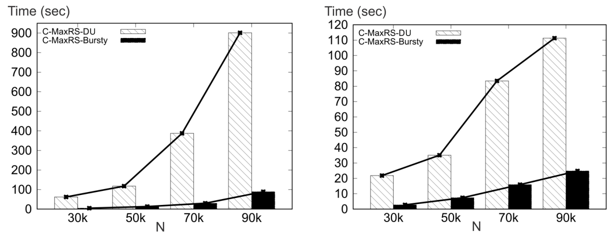


(i) (ii)

Figure 4.17. Varying γ (i) Gaussian (ii) Uniform.

Figure 4.17(i) (uniform distribution), initially when $\gamma = 100$, C-MaxRS-DU (2.99s) performs better than C-MaxRS-Bursty (3.51s). But, for $\gamma = 250$, C-MaxRS-Bursty performs faster, i.e., 7.4s vs 4.88s. Thus, for this setting, there is a value of γ in-between 100 and 250, after which C-MaxRS-Bursty starts out-performing C-MaxRS-DU. This aligns with our intuition that for cases where γ is not too high, C-MaxRS-DU gives us the optimal performance, whereas, C-MaxRS-Bursty is more efficient as γ increases.

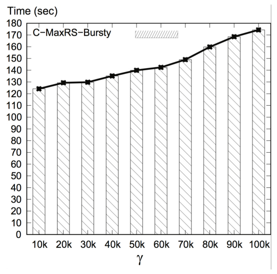
In Figure 4.17, as the value of γ increases, the processing time for C-MaxRS-DU increases exponentially, but the increase in C-MaxRS-Bursty is linear. C-MaxRS-Bursty outperforms C-MaxRS-DU by 5.89 times on average for uniform distribution of data, and by 10.94 times in case of Gaussian distribution of data. This experiment shows the effectiveness of C-MaxRS-Bursty for high streaming data.



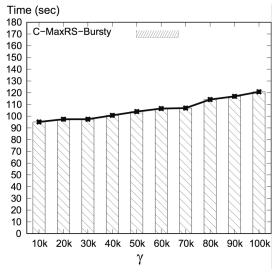
(i) (ii)

Figure 4.18. Varying N (i) Gaussian (ii) Uniform.

4.6.2.3. Varying N**.** Subsequently, we vary the value of N, i.e., number of objects, and preset the results in Figure 4.18. Processing times of both the algorithms increase with the increasing cardinality, although, we note that the increase in C-MaxRS-Bursty is much slower. C-MaxRS-Bursty outperforms C-MaxRS-DU by 5.60 times on average for uniform distribution of data. For Gaussian distribution, C-MaxRS-Bursty outperforms C-MaxRS-DU by 11.34 times on average.



(i)



(ii)

Figure 4.19. γ in larger scale (i) Gaussian (ii) Uniform.

4.6.2.4. Scalability of C-MaxRS-Bursty. In the final experiment, we show the effect of larger γ values on C-MaxRS-Bursty in Figure 4.19. We also use a larger value of N for this experiment – i.e., the value of γ is varied from $10,000$ to $100,000$, and the total number of objects N is set to $200,000$. We omit the performance of C-MaxRS-DU for this experiment as the processing time for large γ values is exponentially high (to avoid skewing the graph). We can see that, the results in Figure 4.19 illustrate similar trend as Figure 4.17, even though we used significantly larger values of γ and N. For both distributions, processing time increases only slightly as the value of γ increases. For example, in Figure 4.19(i), for a 10 times increase of γ value (from 10k to 100k), the processing time only increases by 1.4 times (from 124.2s to 174.3s). Same is true for uniform distribution (cf. Figure 4.19(i)), where this increase is even less (1.27 times, i.e., from 95.1s to 124.8s). We also note that, the bulk of the processing time of C-MaxRS-Bursty is consumed by lines 32 – 33 of Algorithm 13– i.e., executing the function $PrepareSlices$ and $SliceSearchMR$. These results demonstrate the scalability of C-MaxRS-Bursty – where it is ensured that recomputation (i.e., lines 32 – 33) is performed only once (in worst case) instead of γ times.

Algorithm 13 SolveCMaxRSBursty $(E_{new}, a, b, \theta, p_c^*)$

```
Require: New events $E_{new}$, $a \times b$, slice-width constant $\theta$, and current maximal point $p_c^*$
Ensure: Updated maximal point $p_c^*$
 1: $S_{index} \leftarrow$ BuildIndex$(E_{new}, \theta, b)$
 2: $S_{lazy} \leftarrow$ set of slices marked lazy
 3: $isPrunable \leftarrow true$
 4: $prev \leftarrow NULL$
 5: $cur \leftarrow S_{index}.get(1)$
 6: for each $s_i \in S_{slice}$ from left to right $(i = 1, 2, \ldots)$ do
 7:    $next \leftarrow S_{index}.get(i+1)$
 8:    $E_{cur\_slice} \leftarrow prev.E_{rr} \cup next.E_{lr} \cup cur.E_{lr} \cup cur.E_{nr} \cup cur.E_{rr}$
 9:    if $|E_{cur\_slice}| > 0$ then
10:       for each $e^+ \in E_{cur\_slice}$ do
11:          $r_e \leftarrow$ the $a \times b$ rectangle centered at $o_e$
12:          $QTree.insert(r_e)$
13:          $s_i.R \leftarrow s_i.R \cup r_e$
14:       end for
15:       for each $e^- \in E_{cur\_slice}$ do
16:          $r_e \leftarrow$ the $a \times b$ rectangle centered at $o_e$
17:          $QTree.delete(r_e)$
18:          if before the removal $r_e \in A(s_i.p_c^*)$ then
19:             $s_i.lazy \leftarrow true$
20:             $S_{lazy} \leftarrow S_{lazy} \cup \{s_i\}$
21:          end if
22:          $s_i.R \leftarrow s_i.R - r_e$
23:       end for
24:       if atleast one $e^+$ in $E_{cur\_slice}$ and $s_i.R$ conforms to $X$ then
25:          $isPrunable \leftarrow false$
26:          $s_i.lazy \leftarrow true$
27:          $S_{lazy} \leftarrow S_{lazy} \cup \{s_i\}$
28:       end if
29:    end if
30:    $prev \leftarrow cur$
31:    $cur \leftarrow next$
32: end for
33: $s_{max} \leftarrow$slice containg global $p_c^*$
34: if $s_{max}.lazy = true$ or $isPrunable = false$ then
35:    $PrepareSlices(S_{lazy})$
36:    $p_c^* \leftarrow$ SliceSearchMR$(p_c^*)$
37: end if
38: return $p_c^*$
```

CHAPTER 5

MAxRS^3 for Spatial Shapes

In this chapter, we first review the basics of sweep-line technique, and the standard approach for solving MaxRS (revisit) and P-MaxRS problem. We then proceed on to introduce the problem of MAxRS^3 and present a formal definition of the problem. Subsequently, we provide the details of a novel and efficient algorithm to compute MAxRS^3 for a given set of spatial shapes.

5.1. Background: Sweepline Algorithms, P-MaxRS

5.1.1. Sweepline Algorithms

Fundamentally, both the standard MaxRS problem and P-MaxRS problem solutions are based on a sweep-line technique [**68, 10**] – a paradigm of conceptually "sweeping" a horizontal (or vertical) line across the plane, stopping at certain discrete points (called *events*) to perform different tests/computations. The events are marked by corresponding Y-coordinates (for horizontal sweep-line) or X-coordinates (for a vertical sweep-line) at which "something interesting" happens.

At each event $e_i \in E$, some geometric computations need to performed with the objects that either intersect or are in the immediate vicinity of the sweep line, and the final solution is available once the line has passed over all objects.

In general, sweep-line algorithms maintain a data structure to store the events, generally sorted by X or Y coordinates, and at a given instance, the data structure stores only the active events. The overall processing time for a sweep-line algorithm is $O(|E| \times P_{e_i})$, where $|E|$ = total number of events, and P_{e_i} = the processing time of each event. Thus, when designing a sweep-line technique, the goal is to minimize $|E|$ and P_{e_i}.

5.1.2. MaxRS for Point Objects

Let $C(p, R)$ denote the region covered by an isothetic rectangle R, placed at a particular point p. Given a rectangular spatial field $\mathbb{F}$, an axis-parallel rectangle R (of size $d_1 \times d_2$), and a set O of n spatial points $O = \{o_1, o_2, \ldots, o_n\}$ (bounded by $\mathbb{F}$), where each o_i is associated with a weight w_i, the answer to MaxRS query ($\mathbb{A}_{MaxRS}(O, R)$) retrieves a position p for placing the center of R, such that $\sum_{\{o_i \in (\,O \cap C(p,R))\}} w_i$ is maximal. If $\forall o_i \in O : w_i = 1$, we have the *count* variant (cf. Figure 1.4). An in-memory solution to MaxRS (cf. [**58**]) transforms it into a "dual" *rectangle intersection problem* by replacing each object in $o_i \in O$ by a $d_1 \times d_2$ rectangle r_i, centered at o_i. R covers o_i if and only if its center is placed within r_i. Thus, the rectangle covering the maximum number of objects can be centered anywhere within the area containing a maximal number of intersecting dual rectangles .

Using this transformation, [**58**] proposed a sweep-line algorithm to solve the MaxRS problem. Viewing the top and the bottom edges of each rectangle as horizontal intervals, an *interval tree* – i.e., a binary tree on the intervals – is constructed, and then a horizontal line is swept vertically. The line stops at the top and bottom edges of each rectangle (a.k.a. events). During each event ($|E|$=$2n$), the interval tree is updated accordingly, and the count (i.e., the number of overlapping rectangles) for each interval currently residing in the tree is computed (P_e=$O(\log n)$). An interval with the maximum count during the entire process is returned as the final solution and, the algorithm takes $O(n \log n)$ (i.e., $O(|E| \times P_{e_i})$) time.

5.1.3. MaxRS for Polygons

[**58**] proposed an extension that considers polygons instead of point objects. The problem addressed in [**58**] (**(P-MaxRS)**) is: Given a rectangular spatial field $\mathbb{F}$, an axis-parallel rectangle R (of size $d_1 \times d_2$), and a set S of n non-overlapping spatial regions (convex polygons) $S = \{s_1, s_2, \ldots, s_n\}$ (bounded by $\mathbb{F}$), the answer to P-MaxRS query ($\mathbb{A}_{P\text{-}MaxRS}(S, R)$) retrieves a position p for placing the center of R, such that:

$$\sum_{\{\forall s_i \in S\}} \begin{cases} 1, & \text{if } (s_i \cap C(p, R)) = s_i \\ 0, & \text{otherwise} \end{cases}$$

is maximal.

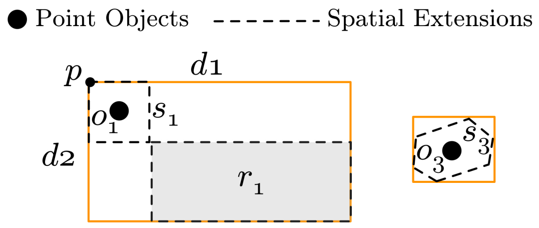


Figure 5.1. P-MaxRS processing scheme.

$(s_i \cap C(p, R)) = s_i$ ensures that s_i is fully enclosed within R. At first, a base solution is devised assuming all $s_i \in S$ are axis-parallel rectangles. If any given s_i is larger than R, it can safely be pruned, i.e., it cannot be fully enclosed by R. For a polygon s_i, we have to place R with its top-left corner at p, where p is the top-left corner of s_i (cf. s_1 in Figure 5.1). Suppose, r_i is the rectangle drawn from the bottom-right point of s_i – e.g., r_1 in Figure 5.1. Clearly, R will enclose s_i completely, if and only if the bottom-right corner of R lies in r_i, which is defined as the *prime rectangle* for s_i. Given the prime rectangles of all axis-parallel rectangles $s_i \in S$, the problem can be converted to the rectangle intersection problem. In case of arbitrary polygons, R encloses a polygon if and only if R encloses its minimum bounding rectangle (MBR) as shown in Figure 5.1 for s_3. Thus, given MBR s'_i for all $s_i \in S$, the same techniques for axis-parallel rectangles can be applied here too.

5.2. Problem Definition: MAxRS^3

In many practical scenarios, maximizing the overall coverage area over the given set of polygons is of more importance than the P-MaxRS problem which can return a placement with many small polygons (see Figure 1.4). For example, suppose the set of given polygons represent flood-affected spatial regions within a state/country. The objective then is to find a way to maximize aid support by reaching to as large amount of flood-affected area as possible. Let us use $A(r)$ to denote the area of a given region r. Based on these observations, we introduce a novel problem Maximizing Area-Range Sum for Spatial Shapes (MAxRS^3) as follows: Given a rectangular spatial field $\mathbb{F}$, an axis-parallel rectangle R (of size $d_1 \times d_2$), and a set S of n non-overlapping spatial regions (convex polygons) $S = \{s_1, s_2, \ldots, s_n\}$ (bounded by $\mathbb{F}$), the answer to MAxRS^3 query ($\mathbb{A}_{MAxRS^3}(S, R)$) retrieves a position p for placing the center of R, such that $\sum_{\{\forall s_i \in S\}} A\left(s_i \cap C(p, R)\right)$ is maximal.

We term $\sum_{\{\forall s_i \in S\}} A\left(s_i \cap C(p, R)\right)$ as the *score* of any position p for MAxRS^3 problem. We note that both P-MaxRS and MAxRS^3 only consider non-overlapping polygons. If there is overlap between two polygons s_i and s_j, we can either: (1) Combine the two polygons into a new single one, e.g., $s_{new} = s_i \cup s_j$; or, (2) Consider three separate disjoint polygons $s_k (= s_i \cap s_j)$, $s_i - s_k$, and $s_j - s_k$. The ideas presented in this paper can be readily extended to include concave (non self-intersecting) polygons, but for brevity we keep the discussion and algorithms limited to convex case. If not mentioned otherwise, "polygon" refers to "convex polygons" throughout the rest of the paper.

5.3. Processing MAxRS^3

We now discuss the challenges relevant to processing MAxRS^3, and devise an efficient algorithm by using a pair of top-to-bottom sweep-lines, accompanied by two left-to-right sweep-lines. Subsequently, we analyze the time-complexity of the offered solution.

5.3.1. Challenges

Although MaxRS and P-MaxRS have efficient $O(n \log n)$ solutions, processing MAxRS^3 poses a different set of challenges:

• Both MaxRS and P-MaxRS can be transformed into rectangle intersection problem. Same is not true for MAxRS^3 since containing the polygons "completely" is not required. The optimal placement of R for MAxRS^3 considers the area to be included to compute the maximal coverage.

• The discrete event-points need to be identified, along with the corresponding processing at each “interesting” points.

5.3.2. Basic Solution

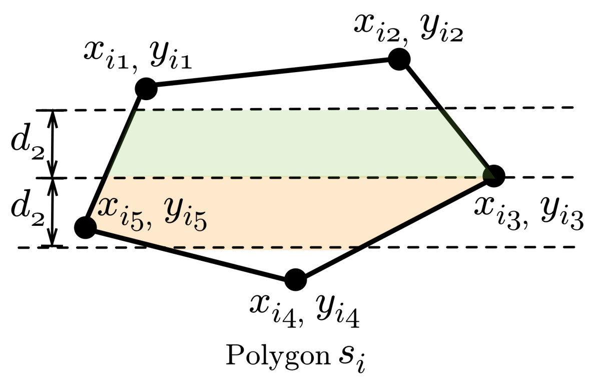


Figure 5.2. Covered area and vertices of a given s_i.

Discrete Points: To identify the events, let us assume that each polygon $s_i \in S$ consists of m_i vertices – $v_{i1}, v_{i2}, \ldots, v_{im_i}$, where $v_{i1} = (x_{i1}, y_{i1})$, $v_{i2} = (x_{i2}, y_{i2})$, ... $v_{im} = (x_{im}, y_{im})$, forming m_i edges by connecting adjacent vertices in a pair-wise manner, e.g., $[\{v_{i1}, v_{i2}\}$, $\{v_{i2}, v_{i3}\}$,..., $\{v_{im_i}, v_{i1}\}]$. From the setting of MAxRS^3, we observe that the area of a polygon covered by R can always be decomposed into a trapezoid (rectangle and squares are a special case of trapezoid) or a triangle. In both cases, the covered area is a function of base (i.e., length of edges) and height – which depends on the slope of certain edges. However, we observe that the slope of any given polygon s_i changes only at the vertices, i.e., $v_{i1}, v_{i2}, \ldots, v_{im_i}$ (see Figure 5.2). Thus, we use vertices of the input polygons as our discrete event-points.

Multiple Sweep-lines: At first, we propose to sweep the space in top-to-bottom manner, i.e., via using a horizontal sweep-line. During each event at a vertex v_{ij}, we have to compute maximal placement of R having the highest coverage in the vicinity of v_{ij}. An interesting observation is that the optimal placement of R may cover both v_{ij}'s above (i.e., up to Y-axis coordinate $(y_{i3} + d_2)$, green in Figure 5.2) or below (i.e., up to Y-axis coordinate $(y_{i3} - d_2)$, orange in Figure 5.2) regions. Thus, we use two sweep-lines in our algorithm, always maintaining a Y-axis distance of d_2 between them. Let us consider an example scenario presented in Figure 5.3, where there are 4 polygons considered: s_1, s_2, s_3 and s_4. The discrete points of interest will be all the vertices v_{ij}, e.g., $v_{11}, v_{21}, v_{31}, v_{41}$, etc. In total, there are 18 such vertices in this setting. At first we will sweep the space in top-to-bottom manner, and use two horizontal sweep-lines: (1) a leader horizontal line (l_h); and (2) a follower horizontal line (f_h). During the whole process, l_h leads (i.e., is below) f_h in the sweeping and the distance between f_h and l_h is set to d_2, i.e., $Y_{l_h} = Y_{f_h} - d_2$ (assuming (0,0) is bottom-left point), where Y_{l_h} and Y_{f_h} denote the Y-coordinate values of the corresponding sweep-lines. Both l_h and f_h stop at all vertices v_{ij} during the sweep – thus, we have two kinds of events: (1) e_{hl}, when the leader horizontal line l_h stops at a vertex; and (2) e_{hf}, when the follower horizontal line f_h stops at a vertex (see Figure 5.3). In total, there will be $18 \times 2 = 36$ events in the example provided in Figure 5.3.

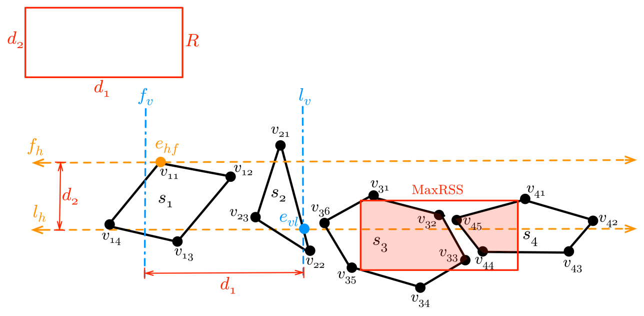


Figure 5.3. Leader and follower sweep-lines for MAxRS^3.

Events Processing Scheme: In case of e_{hl} and e_{hf} events, we will only consider the space bounded by the two horizontal lines l_h and f_h, i.e., $[Y_{l_h}, Y_{f_h}]$. For example, in Figure 5.3, for an event e_{hf} at v_{11}, only the region bounded by the two orange lines will be explored. We will again use the concept of multiple sweep-lines to compute the placement having highest *score* in $[Y_{l_h}, Y_{f_h}]$ bounded region, but this time, sweeping will take place in a left-to-right manner. The idea is to use two vertical sweep-lines: (1) a leader vertical line (l_v); and (2) a follower vertical line (f_v) (blue lines in Figure 5.3). Similar to the horizontal sweep-lines, l_v leads (i.e., is on the right of) f_v and the distance between f_v and l_v is set to d_1, i.e., $X_{l_v} = X_{f_v} + d_1$, where X_{l_v} and X_{f_v} denote the X-coordinate values. For this secondary sweeping process, the discrete points of interest are the current intersection points between the polygons and the horizontal lines – l_h and f_h. Both l_v and f_v stop at all such intersection points during the left-right sweep – thus, we have two kinds of events: (1) e_{vl}, when l_v is involved; and (2) e_{vf}, when f_v is involved. For example, the vertical leader event e_{vl} (blue circle) occurs at one of the intersection points of s_2 and l_h during the horizontal follower event e_{hf}. There will be $11 \times 2 = 22$ such events in the example provided in Figure 5.3.

During each e_{vl} and e_{vf}, we have to find the location p^* having highest *score*, i.e., $\sum_{\{\forall s_i \in S\}} A\left(s_i \cap C(p^*, R)\right)$. For this, we use the fact that the covered area of a given polygon via R can always be decomposed into a trapezoid or triangle. We can compute the area of a given polygon (such as s_i in Figure 5.2, where m_i=5) as follows:

$$A(s_i) = \frac{1}{2} \times \left(\begin{vmatrix} x_{i1} & y_{i1} \\ x_{i2} & y_{i2} \end{vmatrix} + \begin{vmatrix} x_{i2} & y_{i2} \\ x_{i3} & y_{i3} \end{vmatrix} + \ldots + \begin{vmatrix} x_{im_i} & y_{im_i} \\ x_{i1} & y_{i1} \end{vmatrix} \right) \tag{5.1}$$

When performing the left-to-right sweeping:

(1) The intersection points and covered portion of edges of the polygons change. For every little increment of covered portion δ, we already have (or, can pre-compute) the constant slopes of the respective edges. In Equation 5.1, all x_{ij} and y_{ij} values are constants.

(2) We need to maximize A in Equation 5.1 with the optimal placement of R during an event, within the "permissible" ranges of δ. However, given the range for δ, Equation 5.1 is a sum of quadratic functions in δ and its first derivative is a linear one – thus, the extreme can be calculated analytically.

(3) Most importantly, the ranges for δ are always bounded by the event-points of both horizontal and vertical sweep-lines, i.e., e_{hf}, e_{hl}, e_{vf}, and e_{vl}. Thus, we can compute p^* for R during a vertical line-event, even if p^* is in somewhere between two consecutive events.

In summary, we perform a top-to-bottom sweep-line technique using two horizontal lines l_h and f_h, and then, perform a left-to-right sweep of the bounded space by two vertical lines l_v and f_v at each e_{hf} or e_{hl}. During the whole process, we keep track of the maximal coverage area and placement p^*, and eventually, return the result at the end of the top-to-bottom sweeping.

5.4. Algorithmic Details

The processing of MAxRS^3 is formalized in Algorithm 14. In line 1, vertices of all the polygons are sorted in order of their y_{ij} value and inserted into a list v_{list}. In lines 2 – 5, relevant variables are initialized. Lines 6 – 18 constitute the main working loop, i.e., the top-to-bottom sweeping. Lines 7 – 14 check whether the next event should be e_{hl} or e_{hf}, and variables are updated accordingly. Line 15 performs the left-to-right sweeping using l_v and f_v. For brevity, we skip the details of this secondary sweeping in Algorithm 14. The maximal coverage area and optimal placement is tracked via lines 16 – 18.

Algorithm 14 Process-MAxRS^3$(S, R, \mathbb{F})$

```
Require: A set of non-overlapping convex polygons $S$, query rectangle $R$ of size $d_1 \times d_2$
    and bounding box $\mathbb{F}$
Ensure: $\mathbb{A}_{MAxRS^3}$ (i.e., $p^*$)
 1: $v_{list} \leftarrow$ the list of all vertices $v_{ij}$ of each polygon $s_i \in S$ sorted by their $y_{ij}$ value
 2: $Y_{l_h} \leftarrow \mathbb{F}.height$
 3: $Y_{f_h} \leftarrow \mathbb{F}.height + d_2$
 4: $e_{hl}\_index, e_{hf}\_index \leftarrow 0$
 5: $p^*, max\_coverage\_area \leftarrow NULL, 0$
 6: while $e_{hl}\_index < |v_{list}|$ or $e_{hf}\_index < |v_{list}|$ do
 7:     if $(Y_{l_h} - v_{list}[e_{hl}\_index].y_{ij}) \leq (Y_{f_h} - v_{list}[e_{hf}\_index].y_{ij})$ then
 8:         $Y_{l_h} \leftarrow v_{list}[e_{hl}\_index].y_{ij}$
 9:         $Y_{f_h} \leftarrow Y_{l_h} + d_2$
10:         $e_{hl}\_index \leftarrow e_{hl}\_index + 1$
11:     else
12:         $Y_{f_h} \leftarrow v_{list}[e_{hf}\_index].y_{ij}$
13:         $Y_{l_h} \leftarrow Y_{f_h} - d_2$
14:         $e_{hf}\_index \leftarrow e_{hf}\_index + 1$
15:     end if
16:     $p^{local}, local\_coverage\_area \leftarrow$ Perform left-to-right sweep using vertical lines $l_v$ and
        $f_v$
17:     if $local\_coverage\_area > max\_coverage\_area$ then
18:         $max\_coverage\_area \leftarrow local\_coverage\_area$
19:         $p^* \leftarrow p^{local}$
20:     end if
21: end while
22: return $p^*$
```

5.4.1. Time Complexity

Let us assume that there are n polygons, with m vertices each – so sorting in line 1 of Algorithm 14 takes $O((n \times m)\log(n \times m))$. There will be $O(n \times m)$ horizontal line events, i.e., e_{hl} or e_{hf} – (cf. lines 6 – 18 of Algorithm 14). In each such event, when the left-to-right sweep starts, there can be at most $2 \times 2 = 4$ intersections per non-overlapping convex polygons with l_h and f_h, i.e., $O(n)$ intersections in total. The area-calculation needs $O(m)$ number of 2×2 determinants per polygon, taking a worst-case total cost of $O(n \times m)$ at each e_{hl} or e_{hf} event. Thus, the overall time complexity is $O((n \times m)^2)$.

CHAPTER 6

Conclusion, Remaining Work and Future Direction

In this chapter, we first present our ongoing works and expected contributions in the following direction: predicting traffic speed for the densest regions during anomalous events. In the final part, we conclude by briefly discussing the contributions of our already completed works, and then, propose future directions and schedule for completing the remaining works.

6.1. Traffic Prediction at Anomalous Events

Another ongoing work involves the problem of accurate short-term prediction of traffic speed variation in anomalous scenarios. We propose an event-based algorithm that reacts to detection of anomalous events (e.g., New Year's eve, ball games, concerts, etc.) by generating a collection of features that are subsequently used in the prediction model. Specifically, we introduce a novel mobility demand feature which, when combined with other relevant traffic-speed features, turns out to significantly improve the prediction accuracy. The prediction model, in turn, is based on a Multi-Layer Perceptron (MLP) kind of a feedforward artificial neural network. We present experimental observations that demonstrate the improvements in the accuracy of the predicted traffic-speed fluctuations during anomalous events. This is closely related to our discussed MaxRS (and its variants) problem, as we try to find the densest region of mobility requests/Uber calls, and predicting the price of such services during anomalous events. Furthermore, our focus is on predicting traffic speed for densest regions in the whole network.

Traffic prediction is essential for efficient and effective management of any Intelligent Transportation System (ITS) and related categories of services, e.g., routing/navigation, traffic-lights control, dynamic pricing of services in ridesharing platforms like Uber and Lyft [**28**], etc. Many statistical and machine learning models have been proposed to improve prediction accuracy – e.g., spectral analysis, regression methods, time series model, Kalman filtering methods, neural networks, and other hybrid models [**86, 26**]. Traffic models rely on the historical traffic data from various sensing devices, e.g., Global Positioning System (GPS), cameras, etc. – and due to advances in miniaturization of GPS-enabled devices and networking/communication, large volumes of mobility-related dataset are available. This, in turn, has spurred interests in various data mining problems related to urban activities and transportation scenarios. Recently, models have been proposed to indirectly infer and predict traffic condition from heterogeneous data sources (e.g., CO_2 concentration in areas with high buildings density [**89**]).

At the heart of the motivation for this work is the observation that although most of the mobility-related activities in everyday life may be periodic and with predictable patterns, anomalous events may render traffic models invalid, and a fast and effective reaction may be needed to enable catering to different transportation demands.

Example: On the New Year's Eve 2016, an estimated one million people ushered in the new year in Times Square [**5**]. Fig. 6.1 is a visualization for traffic speed of 11th Avenue on Jan 1st, 2016. There is a sudden drop of traffic speed after midnight, which reflects the traffic congestion incurred by one million people's leaving from Times Square.

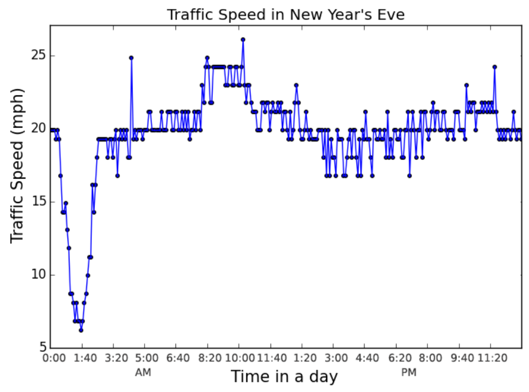


Figure 6.1. Traffic speed of 11th Ave. on Jan 1st, 2016.

For the purpose of this work, *urban anomalies* are defined as occurrences with spatio-temporal extent, underlining abnormal mobility distributions. Urban anomalies result in mobility black holes and volcanoes in Spatio-Temporal Graph (STG) model [**30**] – i.e., generate abnormal traffic flow. Because of their infrequent occurrence and their random nature (in terms of traffic/mobility impact), it is very hard for traditional traffic models to capture the impact of the anomalous change based on historical traffic data only. As hinted in the example above, urban anomalies are often correlated with events whereby people gather and disperse in a short period of time (e.g., ball games, concerts, social events, etc.) – which motivated us to utilize real-time mobility demand (i.e., taxi/Uber/Lyft requests and bike-sharing data) to both detect such events and predict resulting anomalous traffic flow.

Our main contribution is an introduction of a prediction framework based on a feedforward artificial neural network (ANN) that uses both historical traffic data and the (surge in the) real-time people's mobility demand, in order to improve the prediction accuracy when urban anomalies take place.

6.1.1. Related Literature

There are two basic approaches for characterizing the prediction of traffic descriptors (flow, density, speed) in the existing literature: long term and short term. Long term prediction aims to model the physical processes governing the evolution of traffic under normal scenarios [**89**], whereas short term traffic prediction attempts to predict day-to-day, even hour-to-hour status of the traffic [**86**]. Furthermore, the studies of short term traffic prediction can be categorized into two basic types: the first one only considers the traffic data collected by sensors, and the second one takes additional data into consideration (e.g., weather context). Many learning and inference algorithms have been proposed for traffic prediction using traffic sensing data [**86, 26**]. For traffic prediction assisted by additional (multisource) information [**89**], two most-considered categories are weather and holidays. Our work belongs to the category of short-term traffic prediction and its main distinction is the incorporation of the variations of the transportation-demands in the prediction model. We propose a variant of [**81**] (statistical approach based on likelihood ratio test) to detect anomalous events from the transportation demand, and subsequently, prepare the input features for the ANN designed to predict the changes in the traffic descriptor (speed) along road segments.

6.1.2. Problem Formulation

In the sequel, we discuss the context of our problem.

Traffic Descriptor. We rely on traffic flow models [**26**], in which, one can interchangeably use different quantities to describe the traffic state – e.g., flow vs. density (vs. (average) speed). However, in macroscopic level (i.e., locally aggregated values), those quantities are typically related – e.g., one can use (similarly to hydrodynamics) $Q(x,t) = \rho(x,t) * V(x,t)$, where $Q(x,t)$ = traffic flow, $\rho(x,t)$ = traffic density, and $V(x,t)$ = (aggregated) speed at time t. We use aggregated traffic speed as a representation of the flow.

Traffic Speed. For a given road segment rs and a time-stamp t_i, we define its traffic speed S_i as the average traffic flow speed within time interval (t_{i-1}, t_i), where t_{i-1} is the previous data reading time. In practice, traffic speeds are typically averaged over multiple data readings and aggregated up to a common time cadence.

Problem Definition. Let S be the time-series data on traffic speed for road segment rs and D represent the related mobility demand data. In addition, *Demand Feature*, denoted as DF, is extracted from demand data D. We will present details on D and DF in the subsequent sections. Assuming the current time t_i and prediction length γ, the traditional *Short-term Traffic Prediction* problem has the following setting: Given the historical traffic speed data S within time interval [0, t_i] for a road segment rs, develop a traffic prediction model $M(\cdot)$ that outputs the traffic speed at time $t_{i+\gamma}$, i.e., $S_{i+\gamma} = M(S)$. Traditional statistical models will be invalid when anomalous events take place. Hence, the problem addressed in this work, *Short-term Anomalous Traffic Prediction* can be defined as follows: Given the historical traffic speed data S *and* demand features DF within time interval [0, t_i] for a road segment rs, develop an extended traffic prediction model $M^*(\cdot)$ that outputs the traffic speed at time $t_{i+\gamma}$, i.e., $S_{i+\gamma} = M^*(S, DF)$.

We propose an adaptive prediction framework to deal with short-term anomalous traffic speed variations, as illustrated in Fig. 6.2. The framework consists of two parts: *batch training* and *online prediction*. In the training stage, the road network is segmented, and then correlated regions are calculated for a targeted road segment. Subsequently, the historical mobility demand data and traffic speed data are preprocessed and stored when anomalous events are detected. Finally, we extract demand features from mobility demand data and train the anomalous traffic prediction model (i.e., ANN) with demand features *and* historical traffic speed. When conducting online prediction, we first check for anomalous events using real-time mobility demand data and then, if anomaly is detected, predict the traffic speed with anomalous traffic model.

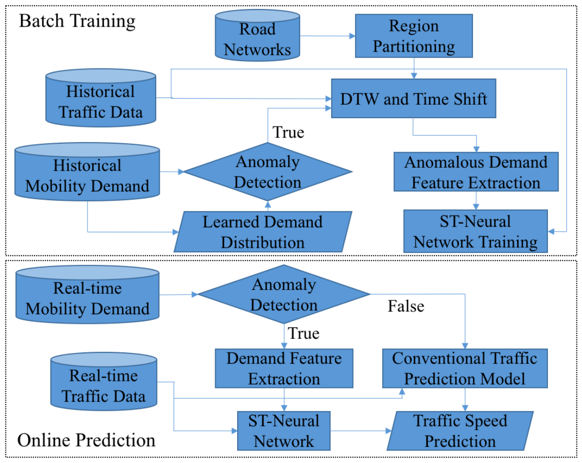


Figure 6.2. Overview of Prediction Framework.

6.2. Conclusion and Contribution

6.2.1. Co-MaxRS

We addressed the problem of determining the locations of a given axes-parallel rectangle R so that the maximum number of moving objects from a given set of trajectories is inside R. In contrast to the MaxRS problem first studied by the computational geometry community [**38, 58**], the Continuous MaxRS (Co-MaxRS) solution may change over time. To avoid checking the validity of the answer-set at every clock-tick, we identified the critical times at which the answer to Co-MaxRS may need to be re-evaluated, corresponding to – events occurring when the dual rectangles of the moving objects change their topological relationship. To speed up the processing of Co-MaxRS we used the kinetic data structures (KDS) paradigm and proposed two pruning heuristics: (1) eliminating events from KDS; and (2) eliminating the objects not affecting the answer (when re-computation of Co-MaxRS is necessary). Moreover, we proposed a hierarchical grid-based indexing scheme, specifically to improve the objects retrieval process (before pruning) for the Co-MaxRS problem. Using the grid-based approach as the base, we also devised an efficient approximation algorithm *Approx-Co-MaxRS* with a worst-case approximation ratio of 4. While our algorithms mostly focused on the moving objects (resp. rectangles) defining the answer set, the possible volume(s) (in terms of 2D space + time) swept by the Co-MaxRS can be straightforwardly derived. Our experiments, over both real and synthetic data sets, showcased that the proposed heuristics enabled significant speed-ups in terms of the overall computation time from the upper bound on the time complexity.

6.2.2. C-MaxRS

In this paper, we have proposed a new variant of MaxRS query, namely *Conditional Maximizing Range-Sum* (C-MaxRS) query in spatial data streams for both non-weighted and weighted objects. Initially, we simply adapted the traditional MaxRS settings to incorporate conditional constraints of different class of objects. However, to handle data streams (i.e., appearance and disappearance of objects) with class-awareness, we needed additional spatial data structures, quadtree and a variant of self-balancing binary tree (e.g., we used AVL-tree), which enabled our algorithm to efficiently compute the changes in the result for different partitions (or slices) of the dataspace. To further improve the overall time-efficiency, we developed two pruning rules: one to handle the appearance of an object and the other to handle disappearance of an object while updating C-MaxRS results. Additionally, to accommodate a different kind of applications settings where a bursty stream of data updates occur in a short time interval, we have proposed a novel technique, C-MaxRS-Bursty to efficiently compute the C-MaxRS results via bulk updates handling. We considered a large parameters space and conducted extensive set of experiments. In sequential spatial data stream scenario, our approach, C-MaxRS-DU yields three to four times improvements (on average) in terms of processing time, when compared to the baseline algorithm. We have also observed that in a bursty scenario, our approach C-MaxRS-Bursty outperforms our one-at-a-time approach, C-MaxRS-DU, by 5-10 times.

6.2.3. MAxRS^3

We introduced a novel variant of the MaxRS problem – the MAxRS^3 problem, which determines the placement for a given fixed rectangle R in a 2D plane, such that the sum of the areas of the intersections of a (subset of a) given collection of polygonal shapes and R is maximized. In many real-world applications, the answers from MAxRS^3 are preferred than other similar problems such as P-MaxRS. We also presented the solution to MAxRS^3 along with the corresponding algorithmic implementation. Finally, we analyzed the time-complexity and processing time for our devised algoritm.

6.3. Possible Future Works and Proposed Schedule

Following are the possible extensions as future works, based on our past and present contributions to the problems:

- While, intuitively, our approaches seem "transferable" to the case of circular Co-MaxRS, we still need to have a more thorough investigation of the pruning effects in the KDS – and a related challenge is to investigate Co-MaxRS when the rectangles are in general positions (i.e., not restricted to be axes-parallel) [**15**].
- In our solution there may be cases where Co-MaxRS has discontinuities – i.e., the current MaxRS needs to instantaneously change its location. Clearly, in practice one may want to have a realistic time-budget for the MaxRS to "travel" from one such location to another – which is another challenge to be addressed, in terms of lost precision.
- Other natural extensions of this setting are to investigate the k-variant of Co-MaxRS – i.e., the case of multiple mobile cameras jointly guaranteeing a continuous maximal coverage, as well as the effective management of Co-MaxRS for real time location updates.
- Similar to the existing parallel computing techniques (e.g., [**69**]), we should also explore the possibility of having a distributed KDS implementation, i.e., dividing the workload between a set of KDS instead of only one.
- We are planning to address a specific aspect of the scalability – namely, distributed data sources over large geographical areas, such as applications in participatory sensing context, at a continent or world-wide scale. In such scenarios, one would naturally want to distribute the computation among multiple regional

servers. A specific angle for this kind of settings is to investigate the trade-offs between different paradigms (e.g., SpatialHadoop [**22**] vs. Spark [**31**]) in a spirit similar to [**83**].

- Another extension is to incorporate the findings from the recent work for monitoring MaxRS over mobile objects [**36**] so that we can optimize mixed-tracking of objects belonging to different categories (e.g., pedestrians, cars, and public transportation users).
- For C-MaxRS, we would like to investigate the trade-offs arising when there is a constraint between the time-instant of a particular update and the update of the answer. This, in some sense, may require a new approach where the bulk update algorithms and data structures proposed in this work will need to be adapted to handle dynamic invocations (e.g., when the buffer of new data reaches certain capacity).
- Complementary to the above, we plan to investigate the C-MaxRS in more traditional streaming settings – i.e., when there is a constraint on the memory and the arrival rate is explicitly taken in consideration. In such cases, relying on data sketches may be inevitable [**18**].
- We are investigating the variations of C-MaxRS where different kinds of mobility may need to be incorporated – for both the users (cf. [**36**]) and the query rectangle (e.g., in the Loon Project settings), as well as the mutual dependencies of both.
- In case of MAxRS^3, we are presently investigating techniques for pruning certain events from consideration during the sweep-line process to speed up the execution.

- We are also working on the scalability aspect of the MAxRS^3 problem – i.e., access structures for the cases when the input is too large to fit in the main memory.
- We are also planning to extend our solution to include spatial objects with extents and 3-D spatial objects.
- In case of traffic prediction for anomalous events (another current/ongoing contribution of us related to finding the densest region in a broader sense), incorporating multi-layer neural network/deep learning into our prediction framework might be a good idea.
- Incorporating different contexts of abnormal demands (for the traffic prediction problem) stemming from different transportation modes – like, for example, the surge of trucks traffic upon arrival/unloading of vessels in ports.
- Again, investigating the benefits of merging multiple regions and their corresponding prediction models could be a possible extension.
- For the problem of MaxRS^3, one practical variation is to consider the possibility of iso contour within the map, i.e., different depths (weights) within single spatial shapes. Instead of discrete, there can also be continuous weight distribution (pdfs).
- Finally, we plan to investigate the impact of the MaxRS problem in *truly* streaming scenarios. The C-MaxRS problem considers a "stream of updates" or "bulk updates". In a truly streaming context, MaxRS is similar to Heavy Hitter Problems [**13**] (in 2-D) for streaming data.

Based on this, following is the tentative schedule we propose to wrap-up the remaining works (in terms of submitting already complete or near-complete projects in Journal/Conference venues):

(1) Submitting the completed extentions of Co-MaxRS problem (approximation and indexing) in ACM TSAS journal. This should be done by November, 2018.

(2) We are working on to devise pruning and index-based extension to the MAxRS^3 problem, and conduct experimental evaluations to compare approaches. This should be done by December, 2018.

(3) Finally, Incoporate deep learning models (e.g., RNN) in the the currently proposed system for traffic speed prediction during anomalous events using mobility demand. Also, we will try to see if incorporating other features such as social media activity has any effect on anomalous events detection and traffic prediction. This should be done by January, 2018.

References

[1] Google Maps API. `https://developers.google.com/maps/`.

[2] OpenWeatherMap Weather API. `http://openweathermap.org/api`.

[3] Vis.js: JavaScript visualization library. `https://visjs.org/`.

[4] Google X Loon Project. `https://x.company/loon/`, 2016. Accessed: 2017-01-31.

[5] What to expect on new years, 2017. https://goo.gl/5dsJhV.

[6] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Comput. Netw.*, 38(4):393–422, Mar. 2002.

[7] D. Amagata and T. Hara. Monitoring MaxRS in spatial data streams. In *19th International Conference on Extending Database Technology, EDBT*, 2016.

[8] G. Anastasi, M. Conti, M. Di Francesco, and A. Passarella. Energy conservation in wireless sensor networks: A survey. *Ad hoc networks*, 7(3):537–568, 2009.

[9] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 350–361. IEEE, 2004.

[10] U. Bartuschka, K. Mehlhorn, and S. Näher. A robust and efficient implementation of a sweep line algorithm for the straight line segment intersection problem. In *IN PROC. WORKSHOP ON ALGORITHM ENGINEERING*. Citeseer, 1997.

[11] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31(1), 1999.

[12] J. Cervino, E. Kalyvianaki, J. Salvachua, and P. Pietzuch. Adaptive provisioning of stream processing systems in the cloud. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 295–301. IEEE, 2012.

[13] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

[14] B. M. Chazelle and D.-T. Lee. On a circle placement problem. *Computing*, 36(1-2), 1986.

[15] Z. Chen, Y. Liu, R. C.-W. Wong, J. Xiong, X. Cheng, and P. Chen. Rotating MaxRS queries. *Information Sciences*, 305, 2015.

[16] D.-W. Choi, C.-W. Chung, and Y. Tao. A scalable algorithm for Maximizing Range Sum in spatial databases. *Proceedings of the VLDB Endowment (PVLDB)*, 5(11), 2012.

[17] D. W. Choi, C. W. Chung, and Y. Tao. Maximizing Range Sum in external memory. *ACM Trans. Database Syst.*, 39(3):21:1–21:44, Oct. 2014.

[18] G. Cormode. Data sketching. *ACM Queue*, 15(2):60, 2017.

[19] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[20] Y. Du, D. Zhang, and T. Xia. The optimal-location query. In *International Symposium on Spatial and Temporal Databases*, pages 163–180. Springer, 2005.

[21] A. Eldawy and M. F. Mokbel. SpatialHadoop: A MapReduce framework for spatial data. In *31st IEEE International Conference on Data Engineering, ICDE*, 2015.

[22] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 1352–1363. IEEE, 2015.

[23] E. Fasolo, M. Rossi, J. Widmer, and M. Zorzi. In-network aggregation techniques for wireless sensor networks: A survey. *Wireless Communications, IEEE*, 14(2):70–87, 2007.

[24] K. Feng, G. Cong, S. S. Bhowmick, W. C. Peng, and C. Miao. Towards best region search for data exploration. In *ACM SIGMOD International Conference on Management of Data*, pages 1055–1070. ACM, 2016.

[25] J. Gudmundsson and M. J. van Kreveld. Computing longest duration flocks in trajectory data. In *ACM GIS Conference*, 2006.

[26] J. Guo, W. Huang, and B. M. Williams. Adaptive kalman filter approach for stochastic short-term traffic flow rate prediction and uncertainty quantification. *Transportation Research Part C: Emerging Technologies*, 43:50–64, 2014.

[27] R. H. Güting and M. Schneider. *Moving objects databases*. Elsevier, 2005.

[28] R. Hahn and R. Metcalfe. The ridesharing revolution: Economic survey and synthesis. In S. D. Kominers and A. Teytelboym, editors, *More Equal by Design: Economic design responses to inequality*. 2017.

[29] F. J. Harvey and K. J. Krizek. Commuter bicyclist behavior and facility disruption. Technical report, 2007.

[30] L. Hong, Y. Zheng, D. Yung, J. Shang, and L. Zou. Detecting urban black holes based on human mobility data. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 35. ACM, 2015.

[31] Z. Huang, Y. Chen, L. Wan, and X. Peng. Geospark sql: An effective framework enabling spatial queries on spark. 6:285, 09 2017.

[32] M. M. Hussain, G. Trajcevski, K. A. Islam, and M. E. Ali. Visualization of range-constrained optimal density clustering of trajectories. In *SSTD*, pages 427–432, 2017.

[33] M. M. Hussain, P. Wongse-ammat, and G. Trajcevski. Demo: Distributed MaxRS in wireless sensor networks. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*. ACM, 2015.

[34] M. M. Hussain, P. Wongse-ammat, and G. Trajcevski. Demo: Distributed MaxRS in wireless sensor networks. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, SenSys '15, pages 479–480. ACM, 2015.

[35] M. M.-u. Hussain, B. Avci, G. Trajcevski, and P. Scheuermann. Incorporating weather updates for public transportation users of recommendation systems. In *Mobile Data Management (MDM), 2016 17th IEEE International Conference on*, volume 1, pages 333–336. IEEE, 2016.

[36] M. M.-u. Hussain, K. A. Islam, G. Trajcevski, and M. E. Ali. Towards efficient maintenance of continuous maxrs query for trajectories. In *20th International Conference on Extending Database Technology, EDBT*, 2017.

[37] M. M.-U. Hussain, G. Trajcevski, K. A. Islam, and M. E. Ali. Visualization of range-constrained optimal density clustering of trajectories. In *International Symposium on Spatial and Temporal Databases*, pages 427–432. Springer, 2017.

[38] 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*, 4(4), 1983.

[39] H. Issa and M. L. Damiani. Efficient access to temporally overlaying spatial and textual trajectories. In *IEEE 17th International Conference on Mobile Data Management, MDM 2016, Porto, Portugal, June 13-16, 2016*, pages 262–271, 2016.

[40] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen. Discovery of convoys in trajectory databases. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1), 2008.

[41] P. Kalnis, N. Mamoulis, and S. Bakiras. On discovering moving clusters in spatio-temporal data. 2005.

[42] I. Kamel and C. Faloutsos. *Parallel R-trees*, volume 21. ACM, 1992.

[43] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation, 2000.

[44] S. Ke, J. Gong, S. Li, Q. Zhu, X. Liu, and Y. Zhang. A hybrid spatio-temporal data indexing method for trajectory databases. *Sensors*, 14(7), 2014.

[45] J. Kim, S.-G. Kim, and B. Nam. Parallel multi-dimensional range query processing with r-trees on gpu. *Journal of Parallel and Distributed Computing*, 73(8):1195–1207, 2013.

[46] J. Kleinberg. Bursty and hierarchical structure in streams. *Data Mining and Knowledge Discovery*, 7(4):373–397, 2003.

[47] M. Koubarakis, T. Sellis, A. Frank, S. Grumbach, R. Güting, C. Jensen, N. Lorentzos, Y. Manolopoulos, E. Nardelli, B. Pernici, H.-J. Scheck, M. Scholl, B. Theodoulidis, and N. Tryfona, editors. *Spatio-Temporal Databases – the CHOROCHRONOS Approach.* Springer-Verlag, 2003.

[48] B. Krishnamachari, D. Estrin, and S. Wicker. The impact of data aggregation in wireless sensor networks. In *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, pages 575–578. IEEE, 2002.

[49] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1):122–173, 2005.

[50] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. Big data: The next frontier for innovation, competition, and productivity, 2011.

[51] M. Mas-ud Hussain, A. Wang, and G. Trajcevski. Co-MaxRS: Continuous Maximizing Range-Sum query. Technical Report NU-EECS-16-08, Dept. of EECS, Northwestern University, 2016.

[52] M. F. Mokbel, L. Alarabi, J. Bao, A. Eldawy, A. Magdy, M. Sarwat, E. Waytas, and S. Yackel. MNTG: An extensible web-based traffic generator. In *Advances in Spatial and Temporal Databases*. 2013.

[53] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-temporal access methods. *IEEE Data Eng. Bull.*, 26(2):40–49, 2003.

[54] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In *ACM SIGMOD International Conference on Management of Data*, 2004.

[55] M. I. Mostafiz, S. Mahmud, M. M.-u. Hussain, M. E. Ali, and G. Trajcevski. Class-based conditional maxrs query in spatial data streams. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, page 13. ACM, 2017.

[56] M. I. Mostafiz, S. M. F. Mahmud, M. M. Hussain, M. E. Ali, and G. Trajcevski. Class-based conditional maxrs query in spatial data streams. In *SSDBM*, 2017.

[57] Y. Nakayama, D. Amagata, and T. Hara. An efficient method for identifying MaxRS location in mobile ad hoc networks. In *Database and Expert Systems Applications - 27th International Conference, DEXA*, 2016.

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

[59] P. M. Narendra and K. Fukunaga. A branch and bound algorithm for feature subset selection. *IEEE Transactions on Computers*, 26(9):917–922, 1977.

[60] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM journal on Computing*, 2(1):33–43, 1973.

[61] B. Ooi, R. Sacks-Davis, and J. Han. Indexing in spatial databases. *Unpublished/Technical Papers*, 1993.

[62] N. Pelekis and Y. Theodoridis. *Mobility Data Management and Exploration.* Springer, 2014.

[63] T.-K. Phan, H. Jung, and U.-M. Kim. An efficient algorithm for Maximizing Range Sum queries in a road network. *The Scientific World Journal*, 2014, 2014.

[64] J. B. Rocha-Junior, A. Vlachou, C. Doulkeridis, and K. Nørvåg. Efficient processing of top-k spatial preference queries. *Proceedings of the VLDB Endowment (PVLDB)*, 4(2), 2010.

[65] D. Romik. Stirling's approximation for n!: The ultimate short proof? *The American Mathematical Monthly*, 107(6):556, 2000.

[66] H. Samet. Applications of spatial data structures. 1990.

[67] L. Sha, P. Lucey, Y. Yue, P. Carr, C. Rohlf, and I. A. Matthews. Chalkboarding: A new spatiotemporal query paradigm for sports play retrieval. In *21st International Conference on Intelligent User Interfaces, IUI*, 2016.

[68] M. I. Shamos and D. Hoey. Geometric intersection problems. In *FOCS*, 1976.

[69] S. B. Shaw and A. Singh. A survey on scheduling and load balancing techniques in cloud computing environment. In *Computer and Communication Technology (ICCT), 2014 International Conference on*, pages 87–95. IEEE, 2014.

[70] S. Shekhar and S. Chawla. *Spatial databases: A tour*, volume 2003. Prentice Hall Upper Saddle River, NJ, 2003.

[71] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010.

[72] D. Šidlauskas, S. Šaltenis, C. W. Christiansen, J. M. Johansen, and D. Šaulys. Trees or grids?: indexing moving objects in main memory. In *Proceedings of the 17th ACM SIGSPATIAL international conference on Advances in Geographic Information Systems*, pages 236–245. ACM, 2009.

[73] Y. Tao, X. Hu, D. Choi, and C. Chung. Approximate MaxRS in spatial databases. *Proceedings of the VLDB Endowment (PVLDB)*, 6(13), 2013.

[74] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In *International Conference on Very Large Data Bases (VLDB)*, 2003.

[75] N. Trigoni and B. Krishnamachari. Sensor network algorithms and applications. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 370(1958):5–10, 2012.

[76] C. R. Vicente, D. Freni, C. Bettini, and C. S. Jensen. Location-related privacy in geo-social networks. *IEEE Internet Computing*, 15(3), 2011.

[77] R. C.-W. Wong, M. T. Özsu, P. S. Yu, A. W.-C. Fu, and L. Liu. Efficient method for maximizing bichromatic reverse nearest neighbor. *Proceedings of the VLDB Endowment (PVLDB)*, 2(1), 2009.

[78] P. Wonge-ammat, M. Mas-ud Hussain, G. Trajcevski, B. Avci, and A. Khokhar. Distributed in-network processing of k-maxrs in wireless sensor networks. In *Proceedings of the 6th International Conference on Sensor Networks (SENSORNETS '17)*, volume 1, pages 108–117. SCITEPRESS, 2017.

[79] P. Wongse-ammat, M. M.-u. Hussain, G. Trajcevski, B. Avci, and A. Khokhar. Distributed in-network processing of k-maxrs in wireless sensor networks. In *7th International Conference on Sensor Networks, SENSORNETS*, 2017.

[80] D. Wu, N. Mamoulis, and J. Shi. Clustering in geo-social networks. *IEEE Data Eng. Bull.*, 38(2), 2015.

[81] M. Wu, X. Song, C. Jermaine, S. Ranka, and J. Gums. A lrt framework for fast spatial anomaly detection. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 887–896. ACM, 2009.

[82] X. Xiao, B. Yao, and F. Li. Optimal location queries in road network databases. In *IEEE International Conference on Data Engineering (ICDE)*, 2011.

[83] S. You, J. Zhang, and L. Gruenwald. Large-scale spatial join query processing in cloud. In *31st IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2015, Seoul, South Korea, April 13-17, 2015*, pages 34–41, 2015.

[84] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *IEEE International Conference on Data Engineering (ICDE)*, 2005.

[85] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *Proceedings of the 2003 ACM SIGMOD*, 2003.

[86] Y. Zhang, Y. Zhang, and A. Haghani. A hybrid short-term traffic flow forecasting method based on spectral analysis and statistical volatility model. *Transportation Research Part C: Emerging Technologies*, 43:65–78, 2014.

[87] Y. Zheng. Trajectory data mining: An overview. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(3), 2015.

[88] Y. Zheng, L. Zhang, X. Xie, and W. Y. Ma. Mining interesting locations and travel sequences from GPS trajectories. In *ACM International Conference on World Wide Web*. ACM, 2009.

[89] Z. Zheng, D. Wang, J. Pei, Y. Yuan, C. Fan, and F. Xiao. Urban traffic prediction through the second use of inexpensive big data from buildings. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1363–1372. ACM, 2016.

[90] Y. Zhong, J. Han, T. Zhang, Z. Li, J. Fang, and G. Chen. Towards parallel spatial query processing for big spatial data. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2085–2094. IEEE, 2012.

[91] X. Zhou, W. Wang, and J. Xu. General purpose index-based method for efficient MaxRS query. In *Database and Expert Systems Applications - 27th International Conference, DEXA*, 2016.

[92] Z. Zhou, W. Wu, X. Li, M. L. Lee, and W. Hsu. MaxFirst for MaxBRkNN. In *IEEE International Conference on Data Engineering (ICDE)*, 2011.

Vita

Muhammed Mas-ud Hussain was born in Dhaka, Bangladesh on December 01, 1988, the son of Md. Liaquat Hussain and Baby Hussain. After completing high school, he entered Bangladesh University of Engineering and Technology, receiving the degree of Bachelor of Science in 2012. After graduation, he worked for two software companies in Dhaka for one and half years, before deciding to pursue Doctoral studies. He joined The Graduate School in the Department of EECS (Electrical Engineering and Computer Science) at Northwestern University in Fall, 2013. In-between, he and Bony Anjabeen tied the knot in the spring of 2013. His research, supervised by Dr. Goce Trajcevski, is focused on *Querying and Mining in Spatial and Moving Objects Databases*, *Applications of Machine Learning Techniques*, *Data Management in Wireless Sensor Networks*, and *Context-Awareness in Recommendation Systems*. In Fall 2018, Mas-ud was appointed as a lecturer by the Department of EECS at Northwestern University, to teach EECS 495: Introduction to Database Systems. He already gained substantial industry research experiences as an intern at Cognitive User Experience Lab in IBM Research (mentor: Mathew Davis), and HERE Research (mentor: Bo Xu). He has also worked as a Machine Learning research intern (mentored by Linbin Yu) at Facebook, and planning to join Facebook as a Research Scientist in 2019. More details on Mas-ud is available here.