Class-based Conditional MaxRS Query in Spatial Data Streams

Mir Imtiaz Mostafiz
Bangladesh University of Engineering and Technology
Dhaka, Bangladesh
1105002.mim@ugrad.cse.buet.ac.bd

S.M.Farabi Mahmud
Bangladesh University of Engineering and Technology
Dhaka, Bangladesh
1105022.sm@ugrad.cse.buet.ac.bd

Muhammed Mas-ud Hussain*
Northwestern University
Dept of EECS, 2145 Sheridan Road
Evanston, Illinois 60208
mas-ud@u.northwestern.edu

Mohammed Eunus Ali
Bangladesh University of Engineering and Technology
Dhaka, Bangladesh
eunus@cse.buet.ac.bd

Goce Trajcevski*
Northwestern University
Dept of EECS, 2145 Sheridan Road
Evanston, Illinois 60208
goce@eecs.northwestern.edu

ABSTRACT

We address the problem of maintaining the correct answer-sets to the *Conditional Maximizing Range-Sum* (C-MaxRS) query in spatial data streams. Given a set of (possibly weighted) 2D point objects, the traditional MaxRS problem determines an optimal placement for an axes-parallel rectangle r so that the number – or, the weighted sum – of objects in its interior is maximized. In many practical settings, the objects from a particular set – e.g., restaurants – can be of distinct types – e.g., fast-food, Asian, etc. The C-MaxRS problem deals with maximizing the overall sum, given class-based existential constraints, i.e., a lower bound on the count of objects of interests from particular classes. We first propose an efficient algorithm to the static C-MaxRS query, and extend the solution to handle dynamic (data streams) settings. Our experiments over datasets of up to 100,000 objects show that the proposed solutions provide significant efficiency benefits.

CCS CONCEPTS

• **Information systems** → **Spatial-temporal systems**; **Location based services**; *Database query processing*;

KEYWORDS

Maximizing Range Sum Query, Constrained Query Processing, Spatial Data Streams, C-MaxRS, Conditional MaxRS

ACM Reference format:
Mir Imtiaz Mostafiz, S.M.Farabi Mahmud, Muhammed Mas-ud Hussain[1], Mohammed Eunus Ali, and Goce Trajcevski[1]. 2017. Class-based Conditional MaxRS Query in Spatial Data Streams. In *Proceedings of SSDBM '17, Chicago, IL, USA, June 27-29, 2017,* 12 pages.
https://doi.org/http://dx.doi.org/10.1145/3085504.3085517

* Research supported by NSF grants III 1213038 and CNS 1646107, ONR grant N00014-14-10215 and HERE grant 30046005.

SSDBM '17, June 27-29, 2017, Chicago, IL, USA

ACM ISBN 978-1-4503-5282-6/17/06…$15.00
https://doi.org/http://dx.doi.org/10.1145/3085504.3085517

1 INTRODUCTION

Rapid advances in accuracy and miniaturization of location-aware devices (e.g., GPS, smartphones, etc.) and increased use of social networks services (e.g., check-in updates) have enabled a generation of large volumes of spatial data [12]. 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 [24, 26].

One particular query that has received recent attention is the *Maximizing Range-Sum (MaxRS)* [5]: 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 [2, 6, 9, 17, 23] have been recently addressed by the spatial database and sensor network communities.

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 different classes: fast-food, Asian, French, etc. Similarly, a vehicle can be a car, a truck, a motor-cycle, and so on. In settings where data can be classified in different sub-categories,

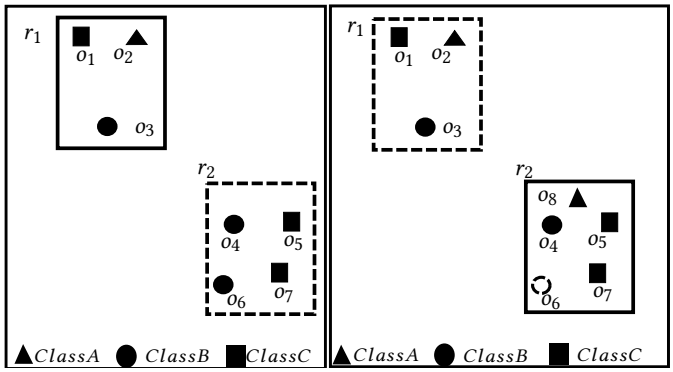


Figure 1: An example of C-MaxRS problem in spatial data streams at time (i) t_1 (ii) t_2 .

there might be class-based existential 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. For example, let us assume that in an area under Google X's **Loon Project** [1], there are different types of users – premium (class A), regular (class B), and free (class C), and users can disconnect or reconnect anytime. Consider the following query:
Q1: *"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 number of users in its coverage is maximized?".*

It is not hard to adapt **Q1** to 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 static scenario, and for dynamic settings we have *Conditional Maximizing Range-Sum for Data Stream* (C-MaxRS-DS) query. An example of C-MaxRS query (for 7 users) with query rectangle size $a \times b$ is shown in Figure 1, with the following conditions: at least 1, 1, and 1 user from classes A, B, and C, respectively. Rectangles r_1 and r_2, with dimension $a \times b$, are two candidates but although r_2 contains most users (the traditional MaxRS solution), r_1 is the solution for the C-MaxRS problem, given the constraints (see Figure 1(i)). Suppose, at t_2, user o_6 disconnects and a new user o_8 joins the system. The C-MaxRS solution will then change to r_2 from r_1 (see Figure 1(ii)).

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 results. The basic processing scheme follows the technique of spatial subdivision from [6], 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, while solving the constrained maximum range sum for the data stream (C-MaxRS-DS) problem. We incorporate 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.
Our main contributions can be summarized as follows:
• We formally define the C-MaxRS problem and provide a baseline solution using spatial subdivision (slices).
• We extend the solution to deal with spatial data streams (appearing and disappearing objects). 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 demonstrate the benefits of our proposed method via experiments over a large dataset.

In the rest of this paper, Section 2 compares the work with respect to the existing literature, while Section 3 formalizes the C-MaxRS problem. Section 4 describes the necessary properties of the conditional weight functions and lays out the basic solution. Section 5 presents the details of our pruning strategies, data structures and algorithms for incorporating dynamic data. Section 6 presents the quantitative experimental analysis and Section 7 summarizes and outlines directions for future work.

2 RELATED WORKS

The Range Aggregation and Maximum Range Sum (MaxRS) queries, and their variants have been extensively studied in recent years [4, 5, 11, 19, 21]. A *Range Aggregation Query*, returning the aggregate result from a set of points, was solved for both 1-dimensional space – i.e. calculating result from set of values in given interval [22] and for 2 dimensional point space, i.e., calculating result from a given rectangle with fixed location [16]. To calculate the aggregate result, an *Aggregate Index*, storing the summarized result for specific region referenced by that index is used [4]. Different data structures are introduced to store the aggregate index – e.g., [11] proposed *Multi-Resolution Aggregate tree* (MRA-tree) to reduce the complexity.

The MaxRS problem was first addressed by researchers in computational geometry [7, 10, 13], based on a technique that finds connected components and a maximum clique of an intersection graph of rectangles in the plane [10]. A solution based on plane sweep strategy was presented in [13], where the input point-objects were "dualized" into rectangles (centered at the points and with dimensions equivalent to the query rectangle r). Then an interval tree was used to record the regions (a.k.a. windows) with highest number of intersecting (dual) rectangles along the sweep – denoting the possible locations for placing the (center of the) query rectangle, yielding $O(n \log n)$ time complexity. However these solutions are not scalable, and [5] proposed scalable extensions suited for LBS-applications – e.g., retrieve best location for a new franchise store with a specified delivery range. Subsequently, different variants of the MaxRS problem have been investigated: – constraining to underlying road networks [17, 25]; – processing MaxRS queries in wireless sensor networks [8, 23]; – considering rotating MaxRS problem [3], where rectangles do not need to be axes parallel allowing much more flexibility. A rather complementary work, tackling the problem of approximate solution to the MaxRS query was presented in [20], using randomized sampling to bound the error with higher probability, with increasing number of objects in question.

Monitoring MaxRS for dynamic settings, where objects can be inserted and/or deleted was first addressed in [2]. To efficiently detect the new locations for placing the query rectangle, [2] exploited the aggregate graph *aG2* in a grid index and devised a branch-and-bound algorithm [14] over that *aG2* graph for efficient approximation. We note that our work is complementary to [2], in the sense that we addressed the settings of having different classes of objects and existential constraints based on them – whereas [2]

solves the basic MaxRS problem. Moreover, [2] 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 [6] – 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 [6] (slices and pruning) – however, we tackle a different context: class-based constraints and dynamic/streaming data updates and, towards that, we also incorporated additional data structures (see Section 5).

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.

3 PRELIMINARIES

We now introduce the C-MaxRS problem, and also extend the definition to include the possibility of appearing/disappearing objects. In addition, we discuss the concept of submodular monotone functions.

***C*-MaxRS & *C*-MaxRS-DS**: 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 is the number of objects of type k_i in the interior of r centered at a point p. A utility function $f(O) : \mathcal{P}(O) \rightarrow \mathbb{N}_0$, 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 3.1. **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 case there is no placement p for which all the conditions of *MinConditionSet* is met, the query will return an empty answer – indicating 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 Stream (C-MaxRS-DS). 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 3.2. **Conditional-MaxRS for Data Stream (C-MaxRS-DS).** 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-DS 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: [6] 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 3.3 (**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) $f(X \cup \{x\}) - f(X) >= f(Y \cup \{x\}) - f(Y)$ 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, 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 we have considered only the counts of POIs when defining the utility function or conditions in X throughout the paper. They can be extended to incorporate different non-negative weights for objects in a straightforward manner – i.e., most of the techniques (including pruning) devised in the work are still applicable with trivial modifications. Similarly, although in our provided examples, for brevity, we've

[1] More formally, [6] was considering submodular monotonic functions as aggregates.

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 BASIC *C*-MAXRS

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

4.1 *C*-MaxRS → 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 [6, 9, 13]) 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 2 using the same example (and conditions) of Figure 1. 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 2, 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 Section 1, while $g(A(p_2))$=$f(\{o_4, o_5, o_6, o_7\}) = 0$ as they do not.

Similarly, C-MaxRS-DS 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:

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

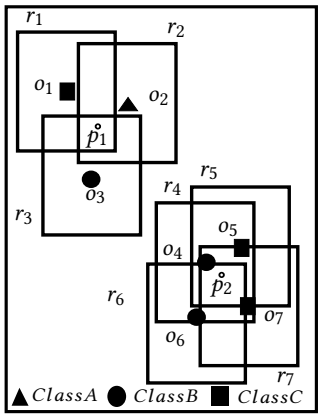


Figure 2: C-MaxRS → dual problem.

where

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

.

4.2 Properties of f and g

[6] devised a method to solve an instance of *Best Region Search* (BRS) problem where the weight function $f : \mathcal{P}(O) \rightarrow \mathbb{R}$ is a submodular monotone function (see Definition 3.3). In [6], 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 4.1. *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 class, if the given 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. But, 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.

□

Lemma 4.2. *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. □

Let us consider the example of Figure 2 – 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.

4.3 Processing C-MaxRS

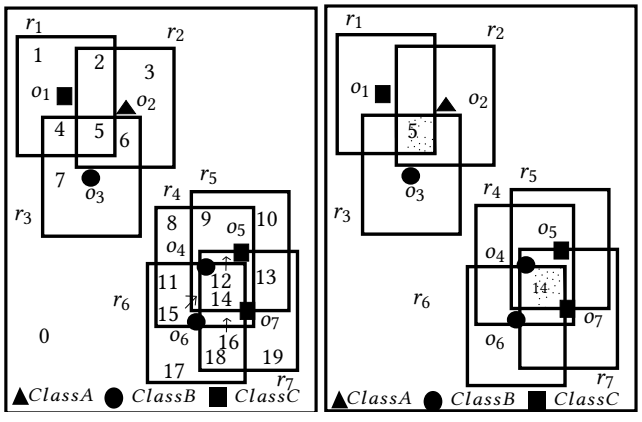
Although f and g are not submodular functions, we show that their monotonicity property can be utilized to derive efficient processing and optimization strategies, similar to the ideas presented in [6].

4.3.1 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 3(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 [6]). 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 3(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 3(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. [6] 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 3), 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).

4.3.2 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 4i, 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 [6], 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



(i) Disjoint regions **(ii) Maximal Regions**

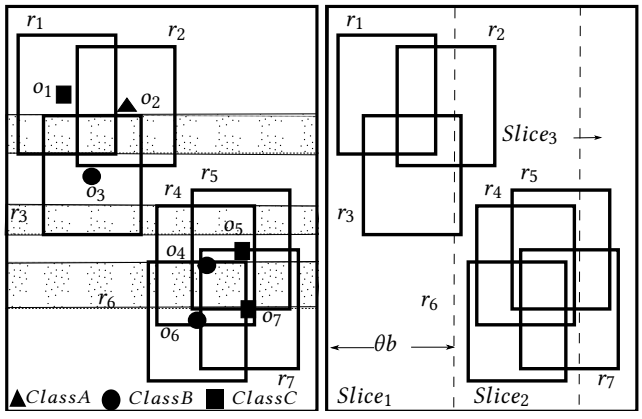
Figure 3: Disjoint & Maximal regions

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 4i, $\{r_4, r_5, r_6, r_7\}$ 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 Section 1).

Finally, the monotonicity of g allows us to adapt another optimization technique introduced in [6] – *slices*. 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 [6]).

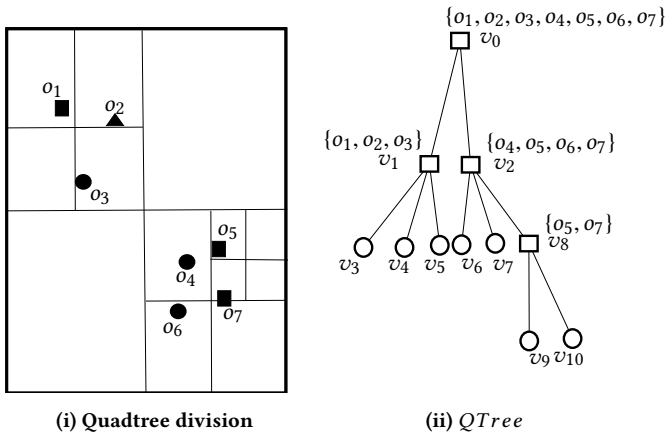
5 C-MAXRS IN DATA STREAMS

Given an efficient solution based on the dual problem and the properties of the utility function, we now proceed to offer 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



(i) Maximal Slabs **(ii) Equal-width Slices**

Figure 4: Maximal Slabs & Slices



(i) Quadtree division **(ii) $QTree$**

Figure 5: Quadtree

C-MaxRS problem (presented in previous section) as the foundation, we augment the solution with compact data-structures and pruning strategies that enable effective handling of data streams environment.

5.1 Data Structures

Before proceeding with the details of the algorithms and pruning schemes, we describe the data structures used. We will 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 [6] 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 [6], we needed self-balancing binary tree to be invoked as dictated by the dynamics of the modifications.

5.1.1 $QTree$. 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 ([18]) – 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 5.

5.1.2 SliceUpperBoundTree. Recall that, the proposed algorithm in Section 4.3 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). On the other hand, there are possibility of appearance (e^+) and disappearance (e^-) events in dynamic streaming scenarios – i.e., upper-bounds of slices (and their respective order) may change frequently with time. To deal with it efficiently, we introduce a balanced binary search tree (*SliceUpperBoundTree*, see [15]) in 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 *SliceUpperBoundTree* 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 *SliceUpperBoundTree* 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 *SliceUpperBoundTree* is much more efficient.

To traverse the slices in decreasing order via *SliceUpperBoundTree*, 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 6, a sample slice structure (of 7 slices) and their respective maximum utility upper bounds (dummy values) are shown for two events at different times 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 6 (ii).

5.1.3 List of Slices. We use a list S_{slice} to maintain 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).

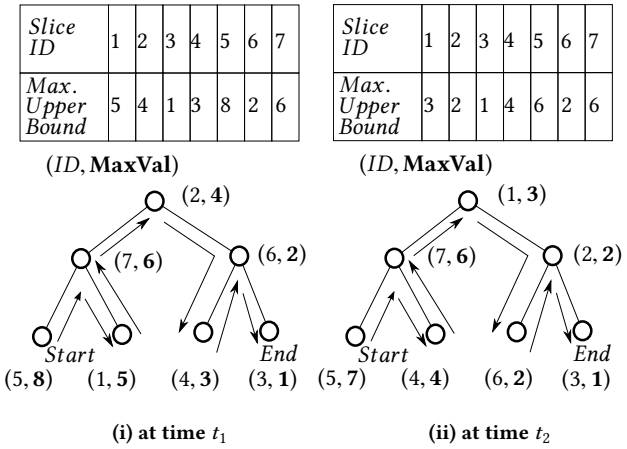


Figure 6: *SliceUpperBoundBST* **at time** t_1 **&** t_2

5.2 Base Method

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

5.2.1 ***PrepareSlices(S_{slice}).*** Function 1 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 4.3.2. *SliceUpperBoundBST* is also build via line 7.

Function 1: PrepareSlices(S_{slice})

Input : 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 $s_i.maxregsearched \leftarrow false$;

5.2.2 ***SliceSearchMR(p_c^*).*** Function 2 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).

5.2.3 ***SolveCMaxRS.*** Algorithm 1 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

Function 2: SliceSearchMR(p_c^*)

Input : Global maximal point p_c^*
Output: Updated global maximal point p_c^*

```
1  c_node ← SliceUpperBoundBST.root;
2  while inorder traversal of SliceUpperBoundBST from
   c_node is not done do
3      if c_node.g_maxub ≤ g(A(p_c^*)) then
4          break;
5      else
6          if S_slice[c_node.slice_i d].maxregsearched = true
           then
7              p_local^* ← S_slice[c_node.slice_i d].p_c
8          else
9              p_local^* ← Compute local optimal point ;
10             S_slice[c_node.slice_i d].p_c ← p_local^*;
11             S_slice[c_node.slice_i d].maxregsearched ← true;
12         if g(A(p_local^*)) > g(A(p_c^*)) then
13             p_c^* ← p_local^*;
14 return p_c^*
```

rectangles in the structure. Line 7 retrieves the list of slices using the given width θb. Finally, the method uses Function 1 to initialize the fields of slices properly in line 8, and computes the C-MaxRS solution using Function 2 in line 9.

Algorithm 1: SolveCMaxRS(O, a, b)

Input : A set of objects O, query size $a \times b$
Output: An optimal point p_c^*

```
1  p_c^* ← null;
2  QTree.init();
3  SliceUpperBoundBST.init();
4  R ← the set of a × b rectangles centered at each o ∈ O;
5  for each r ∈ R do
6      QTree.insert(new Node(r));
7  S_slice ← list of slices of width θb;
8  PrepareSlices(S_slice);
9  p_c^* ← SliceSearchMR(p_c^*);
10 return p_c^*
```

5.3 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 1 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.

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

Lemma 5.1. *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. Let the removed rectangle be 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 4.1, $g(A(p) \setminus \{r_e\}) \leq g(A(p))$. 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 5.1 paves the way for the pruning of slices from being considered a solution at e^- events.

Lemma 5.2. *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 5.1, 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 5.2, we can prune local slice processing at an event e^-, if $r_e \notin A(p_c^*)$, i.e., we need to only update $QTree$ in this case.

Lemma 5.3. *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 5.3 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 5.4. *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 5.1, 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 5.4 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.

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

Lemma 5.5. *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 4.1, $g(A(p) \cup \{r_e\}) \geq g(A(p))$. 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 $\mathbb{F}$. □

For e^- events, we leveraged on ideas like Lemma 5.1 – 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 5.5 – we only process a slice if its affected by the addition of r_e.

Lemma 5.6. *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 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 5.6 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 $diff_i \geq 2$, and if so – we just update $diff_i$ and skip processing that slice. Similarly, if $diff_i \leq 1$, but for $\exists diff_j \geq 1$, 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 3 (although, for brevity, we skip details of implementing and maintaining *diff* in algorithms) .

5.4 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.

5.4.1 SolveCMaxRS$^-$. In Algorithm 2, we present the detailed method for maintaining C-MaxRS result during an e^- event using the ideas introduced in Section 5.3.1. At first r_e is retrieved (from o_e) and then deleted from then *QTree* is updated accordingly (see lines 1-2). 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 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.

5.4.2 SolveCMaxRS$^+$. In Algorithm 3, 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 5.6

Algorithm 2: SolveCMaxRS$^-$ $(e^-(o_e), a, b, p_c^*)$

Input : An $e^-(o_e)$ event, query size $a \times b$, and current maximal point p_c^*
Output: 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 $s_{max} \leftarrow$slice containg global p_c^*;
10 **if** $s_{max}.lazy = true$ **then**
11 PrepareSlices(S_{lazy});
12 $p_c^* \leftarrow$ SliceSearchMR(p_c^*);
13 **return** p_c^*

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 5.6 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).

Algorithm 3: SolveCMaxRS$^+$ $(e^+(o_e), a, b, p_c^*)$

Input : An $e^+(o_e)$ event, query size $a \times b$, and current maximal point p_c^*
Output: 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 **if** $isPrunable = true$ **then**
12 **return** p_c^*
13 PrepareSlices(S_{lazy});
14 $p_c \leftarrow$ SliceSearchMR(p_c^*);
15 **return** p_c^*

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-DS problem, we extended the best known MaxRS solution to cater to C-MaxRS-DS (see Section 5.2 – i.e., processing the C-MaxRS at each event without any pruning) and used it as a baseline.

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 $1000m \times 1000m$ = $1km^2$. 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.

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

Table 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. Note that we exclude the processing time for static C-MaxRS computation 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.

6.1 Performance Evaluation

We now present our detailed observations over different combinations of the parameters.

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 7(i) shows that for Gaussian distribution, the average processing time for our approach (in msec) 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 7(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.

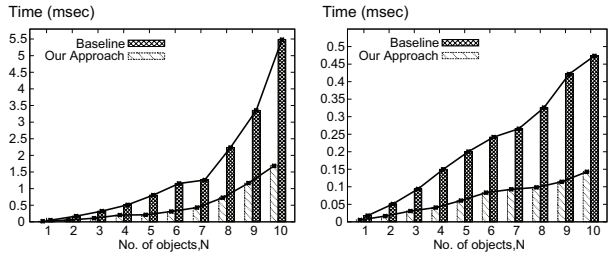


Figure 7: Varying N (i) Gaussian (ii) Uniform.

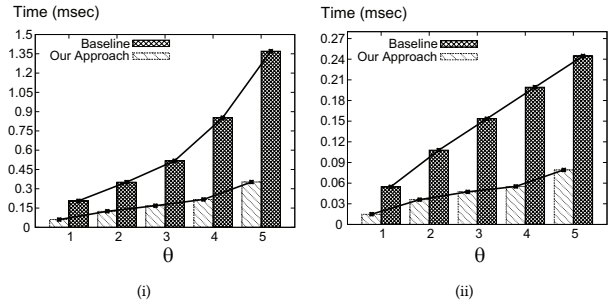


Figure 8: Varying θ (i) Gaussian (ii) Uniform.

6.1.2 Varying Theta (θ). Figure 8 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 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.

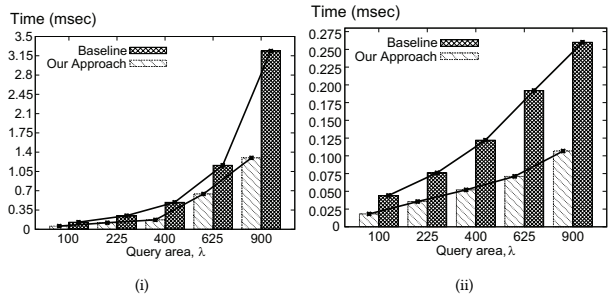


Figure 9: Varying λ (i) Gaussian (ii) Uniform.

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 (in msec) of our approach and baseline algorithm, is shown in Figure 9(i) and Figure 9(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).

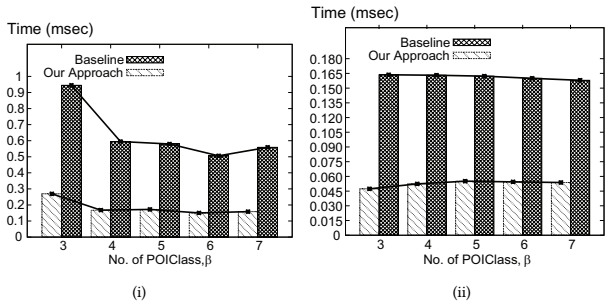


Figure 10: Varying β (i) Gaussian (ii) Uniform.

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 10 (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.

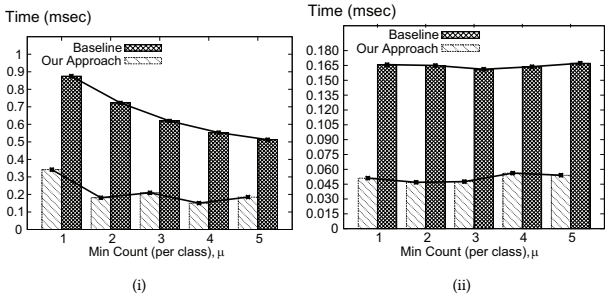


Figure 11: Varying μ (i) Gaussian (ii) Uniform.

6.1.5 Varying Min Class Count, μ. Figure 11 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.

6.1.6 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 stream, 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.

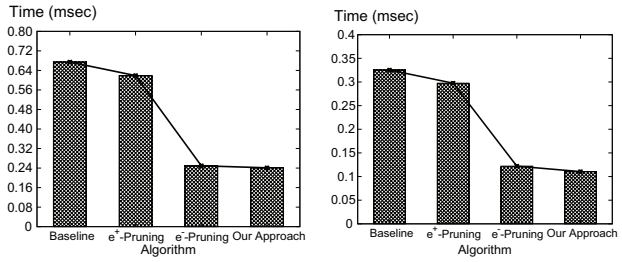


Figure 12: Comparing pruning rules (i) Gaussian (ii) Uniform.

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.

7 CONCLUDING REMARKS AND FUTURE WORK

In this paper, we have proposed a new variant of MaxRS query, namely *Conditional Maximizing Range-Sum* (C-MaxRS) query in spatial data streams. 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. We considered a large parameters space and conducted extensive set of experiments, which demonstrated that our approach yields three to four times improvements (on average) in terms of processing time, when compared to the baseline algorithm.

There are several immediate extensions to our work. Firstly, we are planning to address the scalability aspect – namely, although our experimental results were conducted with a dataset of 100,000 objects, one can foresee scenarios (e.g., obtaining readings in participatory sensing settings at a continent-wide scale) where one may want to distribute the computation among multiple geo-regional servers. Another extension is to incorporate the findings from the recent work for monitoring MaxRS over mobile objects [9] so that we can optimize mixed-tracking of objects belonging to different categories (e.g., pedestrians, cars, and public transportation users). Besides, we plan to extend our solution to include 3-D spatial objects. Lastly, we note that many of the recent works dealing with the variants of the MaxRS problem have not considered the case where spatial objects may have an extent – and this is an avenue that we plan to pursue, as it may be useful in various practical scenarios – e.g., monitoring the agricultural regions with highest areas of soil salinity that can be covered within a limited time-interval.

REFERENCES

[1] 2016. Google X Loon Project. https://x.company/loon/. (2016). Accessed: 2017-01-31.
[2] Daichi Amagata and Takahiro Hara. 2016. Monitoring MaxRS in Spatial Data Streams. In *19th International Conference on Extending Database Technology*.
[3] Zitong Chen, Yubao Liu, Raymond Chi-Wing Wong, Jiamin Xiong, Xiuyuan Cheng, and Peihuan Chen. 2015. Rotating MaxRS queries. *Information Sciences* 305 (2015).
[4] Hyung-Ju Cho and Chin-Wan Chung. 2007. Indexing range sum queries in spatio-temporal databases. *Information and Software Technology* 49, 4 (2007).
[5] D. W. Choi, C. W. Chung, and Y. Tao. 2014. Maximizing Range Sum in External Memory. *ACM Trans. Database Syst.* 39, 3 (Oct. 2014), 21:1–21:44.
[6] Kaiyu Feng, Gao Cong, Sourav S. Bhowmick, Wen-Chih Peng, and Chunyan Miao. 2016. Towards Best Region Search for Data Exploration. In *ACM SIGMOD International Conference on Management of Data*.
[7] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. 1997. *Range queries in OLAP data cubes*. Vol. 26. ACM.
[8] Muhammed Masud Hussain, Panitan Wongse-ammat, and Goce Trajcevski. 2015. Demo: Distributed MaxRS in Wireless Sensor Networks. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*. ACM.
[9] Muhammed Mas-ud Hussain, Kazi Ashik Islam, Goce Trajcevski, and Mohammed Eunus Ali. 2017. Towards Efficient Maintenance of Continuous MaxRS Query for Trajectories. In *20th International Conference on Extending Database Technology, EDBT*.
[10] Hiroshi Imai and Takao Asano. 1983. Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane. *Journal of algorithms* 4, 4 (1983).
[11] Iosif Lazaridis and Sharad Mehrotra. 2001. Progressive approximate aggregate queries with a multi-resolution tree structure. In *ACM SIGMOD Record*, Vol. 30.
[12] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H Byers. 2011. Big data: The next frontier for innovation, competition, and productivity. (2011).
[13] Subhas C Nandy and Bhargab B Bhattacharya. 1995. A unified algorithm for finding maximum and minimum object enclosing rectangles and cuboids. *Computers & Mathematics with Applications* 29, 8 (1995).
[14] Patrenahalli M. Narendra and Keinosuke Fukunaga. 1977. A branch and bound algorithm for feature subset selection. *IEEE Trans. Comput.* 26, 9 (1977), 917–922.
[15] Jürg Nievergelt and Edward M Reingold. 1973. Binary search trees of bounded balance. *SIAM journal on Computing* 2, 1 (1973), 33–43.
[16] Dimitris Papadias, Panos Kalnis, Jun Zhang, and Yufei Tao. 2001. Efficient OLAP operations in spatial data warehouses. In *International Symposium on Spatial and Temporal Databases*. Springer, 443–459.
[17] Tien-Khoi Phan, HaRim Jung, and Ung-Mo Kim. 2014. An Efficient Algorithm for Maximizing Range Sum Queries in a Road Network. *The Scientific World Journal* 2014 (2014).
[18] Hanan Samet. 1990. Applications of spatial data structures. (1990).
[19] Cheng Sheng and Yufei Tao. 2011. New results on two-dimensional orthogonal range aggregation in external memory. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*.
[20] Yufei Tao, Xiaocheng Hu, Dong-Wan Choi, and Chin-Wan Chung. 2013. Approximate MaxRS in spatial databases. *Proceedings of the VLDB Endowment* 6, 13 (2013), 1546–1557.
[21] Yufei Tao and Dimitris Papadias. 2004. Range aggregate processing in spatial databases. *IEEE Transactions on Knowledge and Data Engineering* 16, 12 (2004).
[22] Yufei Tao, Cheng Sheng, Chin-Wan Chung, and Jong-Ryul Lee. 2014. Range aggregation with set selection. *IEEE transactions on knowledge and data engineering* 26, 5 (2014), 1240–1252.
[23] Panitan Wongse-ammat, Muhammed Mas-ud Hussain, Goce Trajcevski, Besim Avci, and Ashfaq Khokhar. 2017. Distributed In-Network Processing of k-MaxRS in Wireless Sensor Networks. In *7th International Conference on Sensor Networks, SENSORNETS*.
[24] Jun Zhang, Manli Zhu, Dimitris Papadias, Yufei Tao, and Dik Lun Lee. 2003. Location-based spatial queries. In *Proceedings of the 2003 ACM SIGMOD*.
[25] Xiaoling Zhou and Wei Wang. 2016. An Index-Based Method for Efficient Maximizing Range Sum Queries in Road Network. In *Australasian Database Conference*. Springer, 95–109.
[26] Zenan Zhou, Wei Wu, Xiaohui Li, Mong Li Lee, and Wynne Hsu. 2011. MaxFirst for MaxBRkNN. In *Proceedings of the 27th IEEE ICDE 2011*. 828–839.