A

Maximizing Range Sum in External Memory

DONG-WAN CHOI, KAIST
CHIN-WAN CHUNG, KAIST
YUFEI TAO, Chinese University of Hong Kong and KAIST

This article studies the *MaxRS* problem in spatial databases. Given a set O of weighted points and a rectangle r of a given size, the goal of the MaxRS problem is to find a location of r such that the sum of the weights of all the points covered by r is maximized. This problem is useful in many location-based services such as finding the best place for a new franchise store with a limited delivery range and finding the hot spot with the largest number of attractions around for a tourist with a limited reachable range. However, the problem has been studied mainly in the theoretical perspective, particularly, in computational geometry. The existing algorithms from the computational geometry community are in-memory algorithms which do not guarantee the scalability. In this article, we propose a scalable external-memory algorithm (*ExactMaxRS*) for the MaxRS problem, which is optimal in terms of the I/O complexity. In addition, we propose an approximation algorithm (*ApproxMaxCRS*) for the *MaxCRS* problem that is a circle version of the MaxRS problem. We prove the correctness and optimality of the ExactMaxRS algorithm along with the approximation bound of the ApproxMaxCRS algorithm.

Furthermore, motivated by the fact that all the existing solutions simply assume that there is no *tied* area for the best location, we extend the MaxRS problem to a more fundamental problem, namely *AllMaxRS*, so that all the locations with the same best score can be retrieved. We first prove that the AllMaxRS problem cannot be trivially solved by applying the techniques for the MaxRS problem. Then we propose an output-sensitive external-memory algorithm (*TwoPhaseMaxRS*), that gives the exact solution for the AllMaxRS problem through two phases. Also, we prove both the soundness and completeness of the result returned from TwoPhaseMaxRS.

From extensive experimental results, we show that ExactMaxRS and ApproxMaxCRS are *several orders of magnitude* faster than methods adapted from existing algorithms, the approximation bound in practice is much better than the theoretical bound of ApproxMaxCRS, and TwoPhaseMaxRS is not only much faster but also more robust than the straightforward extension of ExactMaxRS.

CCS Concepts: •**Theory of computation** → **Data structures and algorithms for data management;** *Database query processing and optimization (theory);* •**Information systems** → *Spatial-temporal systems;*

General Terms: Algorithms, Theory, Experimentation

Additional Key Words and Phrases: External memory, optimal-location query, range sum, spatial databases

ACM Reference Format:
Dong-Wan Choi, Chin-Wan Chung, and Yufei Tao. 2014. Maximizing Range Sum in External Memory. *ACM Trans. Datab. Syst.* V, N, Article A (January YYYY), 45 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

Author's addresses: D.W. Choi, Department of Computer Science, KAIST, Daejeon, Korea; email: dongwan@islab.kaist.ac.kr; C.W. Chung (corresponding author), Department of Computer Science and Division of Web Science and Technology, KAIST, Daejeon, Korea; email: chungcw@kaist.edu; Y. Tao, Department of Computer Science, Chinese University of Hong Kong, Sha Tin, Hong Kong and Division of Web Science and Technology, KAIST, Daejeon, Korea; email: taoyf@cse.cuhk.ed.hk.

DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

In the era of mobile devices, *location-based service*s (*LBS*s) are being used in a variety of contexts such as emergency, navigation, and tour planning. Essentially, these applications require managing and processing a large amount of location information, and technologies studied in spatial databases are getting a great deal of attention for this purpose. Traditional researches in spatial databases, however, have mostly focused on retrieving objects (e.g., *range search*, *nearest neighbor search*, etc.), rather than finding the best location to optimize a certain objective.

Recently, several *location selection problem*s [Du et al. 2005; Rocha-Junior et al. 2010; Wong et al. 2009; Xia et al. 2005; Xiao et al. 2011; Yiu et al. 2007; Zhang et al. 2006; Zhou et al. 2011] have been proposed. One type of these problems is to find a location for a new facility by applying the well-known *facility location problem* in theory to database problems such as *optimal-location queries* and *bichromatic reverse nearest neighbor queries*. Another type of location selection problems is to choose one of the predefined candidate locations based on a given ranking function such as *spatial preference queries*.

In this article, we solve the *maximizing range sum* (*MaxRS*) *problem* in spatial databases. Given a set O of weighted points (a.k.a. objects) and a rectangle r of a given size, the goal of the MaxRS problem is to find a location of r which maximizes the sum of the weights of all the objects covered by r. Figure 1 shows an instance of the MaxRS problem where the size of r is specified as $d_1 \times d_2$. In this example, if we assume that the weights of all the objects are equally set to 1, the center point of the rectangle in solid line is the solution, since it covers the largest number of objects which is 8. The figure also shows some other positions for r, but it should be noted that there are *infinitely many* such positions – r can be anywhere in the data space. The MaxRS problem is different from existing location selection problems mentioned earlier in that there are no predefined candidate locations or other facilities to compete with. Furthermore, this problem is also different from *range aggregate queries* [Sheng and Tao 2011] in the sense that we do not have a known query rectangle, but rather, must discover the best rectangle in the data space.

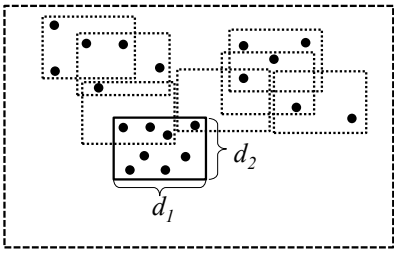


Fig. 1. An instance of the MaxRS problem

An intuitive application of MaxRS is related to many kinds of facilities that should be associated with a region of a certain size. For example, if we open a new pizza franchise store that has a limited delivery range in a downtown area, it is important to maximize the number of residents in a rectangular area[1] around the pizza store. This case is about finding a more profitable place to set up a new service facility.

[1]Note that a range in a downtown area is well represented as a rectangle because an urban area is widely modelled as a grid of city blocks and streets.

For an opposite case, the MaxRS problem can be applied to find a more serviceable place for mobile users. Consider a tourist who wants to find the most representative spot in a city. In this case, the tourist will prefer to visit as many attractions as possible around the spot, and at the same time s/he usually does not want to go too far away from the spot.

In addition, MaxRS can also play an important role in spatial data mining where various kinds of massive location log datasets are employed. Indeed, many works are being reported to mine interesting locations from a large amount of GPS log data of mobile subscribers [Srivastava et al. 2011; Tiwari and Kaushik 2012; Zheng et al. 2009]. These works are mostly involved in extracting the *hot spot* from a massive dataset of points, which can be naturally abstracted as MaxRS. Also, some location datasets are associated with a set of events such as the traffic accident and the crime. For instance, when we do the geographic profiling in crime analysis, it is frequently required to determine the most probable area of offender residence by analysing the set of crime locations [2]. This task is also strongly related to the MaxRS problem.

There has been little research for this natural problem in the database community. In fact, this problem has been mainly studied in the computational geometry community. The first optimal in-memory algorithm for finding the position of a fixed-size rectangle enclosing the maximum number of points was proposed in [Imai and Asano 1983]. Later, a solution to the problem of finding the position of a fixed-size circle enclosing the maximum number of points was provided in [Chazelle and Lee 1986].

Unfortunately, these in-memory algorithms are not scalable for processing a massive dataset of spatial objects such as a GPS log dataset of mobile users and a set of crime locations for several years, since they are developed based on the assumption that the entire dataset can be loaded in the main memory. A straightforward adaptation of these in-memory algorithms into the external memory can be considerably inefficient due to the occurrence of excessive I/O's.

In this article, we propose the first external-memory algorithm, called *ExactMaxRS*, for the maximizing range sum (MaxRS) problem. The basic processing scheme of ExactMaxRS follows the *distribution-sweep* paradigm [Goodrich et al. 1993], which was introduced as an external version of the *plane-sweep* algorithm. Basically, we divide the entire dataset into smaller sets, and recursively process the smaller datasets until the size of a dataset gets small enough to fit in memory. By doing this, the ExactMaxRS algorithm gives an exact solution to the MaxRS problem. We derive the upper bound of the I/O complexity of the algorithm. Indeed, this upper bound is proved to be the lower bound under the comparison model in external memory, which implies that our algorithm is optimal.

In addition, we propose an approximation algorithm, called *ApproxMaxCRS*, for the *maximizing circular range sum* (*MaxCRS*) *problem*. This problem is the circle version of the MaxRS problem, and is more useful than the rectangle version, when a boundary with the same distance from a location is required. In order to solve the MaxCRS problem, we apply the ExactMaxRS algorithm to the set of Minimum Bounding Rectangles (MBR) of the data circles. After obtaining a solution from the ExactMaxRS algorithm, we find an approximate solution for the MaxCRS problem by choosing one of the candidate points, which are generated from the point returned from the ExactMaxRS algorithm. We prove that ApproxMaxCRS gives a (1/4)-approximate solution in the worst case, and also show by experiments that the approximation ratio is much better in practice.

As a major extension of the MaxRS problem, this article also propose a new fundamental problem in spatial databases, namely *AllMaxRS*. Given O and r, the goal of

[2]http://en.wikipedia.org/wiki/Geographic_profiling

the AllMaxRS problem is to retrieve all the locations of r which equally have the maximum total weight of objects covered by r. This problem is addressed by the fact that there exist multiple locations having the same best score in many real world LBSs. In this kind of LBSs, it is more desirable to provide users with all these options for the best location rather than recommending an arbitrary location among all those *tied* best locations. To the best of our knowledge, the AllMaxRS problem has not been studied even in theoretical communities.

Unfortunately, it is not trivial at all to solve the AllMaxRS problem by extending the ExactMaxRS algorithm for the MaxRS problem. We prove that the simple extended algorithm is inefficient in terms of the I/O complexity, especially because it is not output-sensitive. This is based on the theoretical analysis on the upper bound of the number of disjoint areas containing all the best locations.

Our approach for the AllMaxRS problem is to divide the entire process into two phases, and thereby we propose a two-phase external-memory algorithm, namely *TwoPhaseMaxRS*. In the first phase, we build the preliminary structures, which tell the information of all the best locations to be finally returned, in a manner similar to the ExactMaxRS algorithm for the MaxRS problem. By means of these preliminary structures, in the second phase, we actually construct and retrieve all the best locations. We prove that the I/O complexity of TwoPhaseMaxRS is *output-sensitive*, and the result returned from TwoPhaseMaxRS is sound and complete.

Contributions.
We summarize our main contributions as follows:

— We propose the ExactMaxRS algorithm, the first external-memory algorithm for the MaxRS problem. We also prove both the correctness and optimality of the algorithm.
— We propose the ApproxMaxCRS algorithm, an approximation algorithm for the MaxCRS problem. We also prove the correctness as well as tightness of the approximation bound with regard to this algorithm.
— We propose a new fundamental problem in spatial databases (as well as in computational geometry), namely the AllMaxRS problem, which is a more general version of the MaxRS problem.
— We propose the TwoPhaseMaxRS algorithm for the AllMaxRS problem and prove that it has an output-sensitive I/O complexity. We also prove both the soundness and completeness of the result returned from TwoPhaseMaxRS.
— We derive the tight upper bound of the number of disjoint areas that contain all the best locations for the AllMaxRS problem, which is of independent interest in the theoretical perspective.
— We experimentally evaluate our algorithms using both real and synthetic datasets. From the experimental results, we show that (1) ExactMaxRS is two orders of magnitude faster than methods adapted from existing algorithms, (2) ApproxMaxCRS is also one or more orders of magnitude faster than its competitor adapted from the *state-of-the-art* algorithm, (3) the approximation bound of ApproxMaxCRS in practice is much better than its theoretical bound, (4) TwoPhaseMaxRS is much faster than the simple extension of ExactMaxRS, and (5) the performance of TwoPhaseMaxRS is more stable than that of the simple extension when varying the values of experimental parameters.

Organization.
In Section 2, we formally define the problems studied in this article, and explain our computation model. In Section 3, related work is discussed. In Section 4, we review the in-memory algorithms proposed in the computational geometry community. In Sections 5 and 6, the ExactMaxRS algorithm and ApproxMaxCRS algorithm are derived,

respectively. The AllMaxRS problem and its TwoPhaseMaxRS algorithm are presented in Sections 7 and 8. In Section 9, we show experimental results. Conclusions are made and future work is discussed in Section 10.

2. PROBLEM FORMULATION

We consider a set of spatial objects, denoted by O. Each object $o \in O$ is located at a point in the 2-dimensional space, and has a non-negative *weight* $w(o)$. We also use P to denote the infinite set of points in the entire data space.

Let $r(p)$ be a rectangular region of a given size centered at a point $p \in P$, and $O_{r(p)}$ be the set of objects covered by $r(p)$. Then the maximizing range sum (MaxRS) problem is formally defined as follows:

Definition 1 (*MaxRS Problem*). Given P, O, and a rectangle of a given size, find a location p that maximizes:

$$\sum_{o \in O_{r(p)}} w(o).$$

Similarly, let $c(p)$ be a circular region centered at p with a given diameter, and $O_{c(p)}$ be the set of objects covered by $c(p)$. Then we define the maximizing circular range sum (MaxCRS) problem as follows:

Definition 2 (*MaxCRS Problem*). Given P, O, and a circle of a given diameter, find a location p that maximizes:

$$\sum_{o \in O_{c(p)}} w(o).$$

For simplicity, we discuss only the SUM function in this article, even though our algorithms can be applied to other aggregates such as COUNT and AVERAGE. Without loss of generality, objects on the boundary of the rectangle or the circle are excluded.

Since we focus on a massive number of objects that do not fit in the main memory, the whole dataset O is assumed to be stored in external memory such as a disk. Therefore, we follow the standard external memory (EM) model [Goodrich et al. 1993] to develop and analyze our algorithms. According to the EM model, we use the following parameters:

N : the number of objects in the database (i.e., $|O|$)
M : the number of objects that can fit in the main memory
B : the number of objects per block

We comply with the assumption that N is much larger than M and B, and the main memory has at least two blocks (i.e., $M \geq 2B$).

In the EM model, the time of an algorithm is measured by the number of I/O's rather than the number of basic operations as in the random access memory (RAM) model. Thus, when we say linear time in the EM model, it means that the number of blocks transferred between the disk and memory is bounded by $O(N/B)$ instead of $O(N)$. Our goal is to minimize the total number of I/O's in our algorithms.

For clarity, Table I summarizes the main symbols that will frequently appear throughout the article.

3. RELATED WORK

We first review the range aggregate processing methods in spatial databases. The range aggregate (RA) query was proposed for the scenario where users are interested

Table I. List of frequent symbols

Symbol	Description
O, R, and C	the set of objects, the set of rectangles, and the set of circles
$o \in O$, $r \in R$, and $c \in C$	an object in O, a rectangle in R, and a circle in C
$w(o)$	the weight of the object o
N	the total number of objects, i.e., $N = \|O\| = \|R\|$
M	the number of objects that can fit in the main memory
B	the number of objects per block
m	the number of slabs at each recursion
γ and γ_i ($i \in [1, m]$)	a slab and its i-th sub-slab
S and S_i ($i \in [1, m]$)	the slab-file of γ and the slab-file of γ_i
X and X_i ($i \in [1, m]$)	the region-file of γ and the region-file of γ_i
t and t_i ($i \in [1, m]$)	a tuple (max-intervals or h-lines) in S and a tuple in S_i
u and u_i ($i \in [1, m]$)	a tuple (max-ranges) in X and a tuple in X_i
ρ	the max-region with regard to R

in summarized information about objects in a given range rather than individual objects. Thus, a RA query returns an aggregation value over objects qualified for a given range. In order to efficiently process RA queries, usually *aggregate index*es [Cho and Chung 2007; Jürgens and Lenz 1998; Lazaridis and Mehrotra 2001; Papadias et al. 2001; Sheng and Tao 2011] are deployed as the underlying access method. To calculate the aggregate value of a query region, a common idea is to store a pre-calculated value for each entry in the index, which usually indicates the aggregation of the region specified by the entry. However, the MaxRS problem cannot be trivially solved by the RA query processing scheme even if the aggregate index could help to reduce the search space, because the key is to find out *where* the best rectangle is. A naive solution to the MaxRS problem is to issue an infinite number of RA queries, which is prohibitively expensive.

Recently, researches about the selection of optimal locations in spatial databases have been reported, and they are the previous work most related to ours. Du et al. proposed the *optimal-location query* [Du et al. 2005], which returns a location in a query region to maximize the *influence* that is defined to be the total weight of the reverse nearest neighbors. They also defined a different query semantics in their extension [Zhang et al. 2006], called *min-dist optimal-location query*. In both works, their problems are stated under L_1 distance. Similarly, the *maximizing bichromatic nearest neighbor* (*MaxBRNN*) *problem* was studied by Wong et al. [Wong et al. 2009] and Zhou et al. [Zhou et al. 2011]. This is similar to the problem in [Du et al. 2005] except that L_2 distance, instead of L_1 distance, is considered, making the problem more difficult. Moreover, Xiao et al. [Xiao et al. 2011] applied optimal-location queries to road network environments.

However, all these works share the spirit of the classic facility location problem, where there are two kinds of objects such as customers and service sites. The goal of these works is essentially to find a location that is far from the competitors and yet close to customers. This is different from the MaxRS (MaxCRS) problem, since we aim at finding a location with the maximum number of objects around, without considering any competitors. We have seen the usefulness of this configuration in Section 1.

There is another type of location selection problems, where the goal is to find top-k spatial sites based on a given ranking function such as the weight of the nearest neighbor. Xia et al. proposed the *top-t most influential site query* [Xia et al. 2005]. Later, the *top-k spatial preference query* was proposed in [Rocha-Junior et al. 2010; Yiu et al. 2007], which deals with a set of classified feature objects such as hotels, restaurants, and markets by extending the previous work. Even though some of these works consider the range sum function as a ranking function, their goal is to choose one of the candidate locations that are predefined. However, there are an infinite number of can-

didate locations in the MaxRS (MaxCRS) problem, which implies that these algorithms are not applicable to the problem we are focusing on.

In the theoretical perspective, MaxRS and MaxCRS have been studied in the past. Specifically, in the computational geometry community, there were researches [Barequet et al. 1997; Dickerson and Scharstein 1998] for the *max-enclosing polygon problem*. The purpose is to find a position of a given polygon to enclose the maximum number of points. This is almost the same as the MaxRS problem, when a polygon is a rectangle. For the *max-enclosing rectangle problem*, Imai et al. proposed an optimal in-memory algorithm [Imai and Asano 1983] whose time complexity is $O(n \log n)$, where n is the number of rectangles. Actually, they solved a problem of finding the maximum clique in the rectangle intersection graph based on the well-known plane-sweep algorithm, which can be also used to solve the max-enclosing rectangle problem by means of a simple transformation [Nandy and Bhattacharya 1995]. Inherently, however, these in-memory algorithms do not consider a scalable environment that we are focusing on.

In company with the above works, there were also works to solve the *max-enclosing circle problem*, which is similar to the MaxCRS problem. Chazelle et al. [Chazelle and Lee 1986] were the first to propose an $O(n^2)$ algorithm for this problem by finding a maximum clique in a circle intersection graph. The max-enclosing circle problem is actually known to be *3*SUM*-hard* [Aronov and Har-Peled 2005], namely, it is widely conjectured that no algorithm can terminate in less than $\Omega(n^2)$ time in the worst case. Therefore, several approximation approaches were proposed to reduce the time complexity. Recently, Berg et al. proposed a $(1 - \epsilon)$-approximation algorithm [Berg et al. 2009] with time complexity $O(n \log n + n\epsilon^{-3})$. They divide the entire dataset into a grid, and then compute the local optimal solution for a grid cell. After that the local solutions of cells are combined using a dynamic-programming scheme. However, it is generally known that a standard implementation of dynamic programming leads to poor I/O performance [Chowdhury and Ramachandran 2006], which is the reason why it is difficult for this algorithm to be scalable.

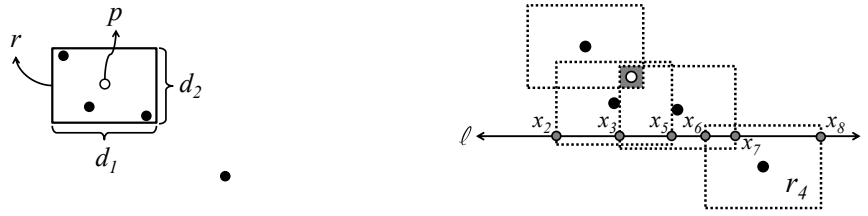
Finally, a preliminary version of this article was published in [Choi et al. 2012]. Even though a scalable solution for the MaxRS (and corresponding MaxCRS) problem is presented in that previous work, the solution considers only a single best location. Here in this long version of the article, by further extending the MaxRS problem, we also tackle a new fundamental problem in spatial databases, namely AllMaxRS, which aims to retrieve all the best locations with the same worth. More specifically, we (1) present the theoretical analysis of the number of the tied best locations, (2) provide an output-sensitive algorithm for the AllMaxRS problem with proofs of its efficiency and correctness, and (3) evaluate the algorithm by conducting extensive experiments.

4. PRELIMINARIES

In this section, we explain more details about the solutions proposed in the computational geometry community. Our solution also shares some of the ideas behind those works. In addition, we show that the existing solutions cannot be easily adapted to our environment, where a massive size of data is considered.

First, let us review the idea of transforming the *max-enclosing rectangle problem* into the *rectangle intersection problem* in [Nandy and Bhattacharya 1995]. The max-enclosing rectangle problem is the same as the MaxRS problem except that it considers only the count of the objects covered by a rectangle (equivalently, each object has weight 1). The rectangle intersection problem is defined as *"Given a set of rectangles, find an area where the most rectangles intersect"*. Even though these two problems appear to be different at first glance, it has been proved that the max-enclosing rectangle problem can be mapped to the rectangle intersection problem [Nandy and Bhattacharya 1995].

We explain this by introducing a mapping example shown in Figure 2. Suppose that the dataset has four objects (black-filled) as shown in Figure 2(a). Given a rectangle of size $d_1 \times d_2$, an optimal point can be the center point p of rectangle r (see Figure 2(a)). To transform the problem, we draw a rectangle of the same size centered at the location of each object as shown in Figure 2(b). It is not difficult to observe that the optimal point p in the max-enclosing rectangle problem can be any point in the most overlapped area (gray-filled) which is the outcome of the rectangle intersection problem. Thus, once we have found the most overlapped area in the transformed rectangle intersection problem, the optimal location of the max-enclosing rectangle problem can trivially be obtained.



(a) Max-enclosing rectangle problem (b) Rectangle intersection problem

Fig. 2. An example of transformation

For the rectangle intersection problem, an *in-memory* algorithm was proposed in [Imai and Asano 1983], which is based on the well-known *plane-sweep* algorithm. Basically, the algorithm regards the top and bottom edges of rectangles as horizontal intervals, and maintains a binary tree on the intervals while sweeping a conceptual horizontal line from bottom to top. For example, there will be 7 intervals in the binary tree when the sweeping line is at ℓ in Figure 2(b), namely $[-\infty, x_2]$, $[x_2, x_3]$, $[x_3, x_5]$, $[x_5, x_6]$, $[x_6, x_7]$, $[x_7, x_8]$, and $[x_8, \infty]$. When the line meets the bottom (top) edge of a rectangle, a corresponding interval is inserted to (deleted from) the binary tree, along with updating the *count*s of intervals currently residing in the tree, where the count of an interval indicates the number of intersecting rectangles within the interval. In Figure 2(b), when the line encounters the top edge of the rectangle r_4, the intervals in the range of $[x_6, x_8]$ will be updated or deleted from the binary tree. Thereafter, the resulting intervals in the binary tree will be $[-\infty, x_2]$, $[x_2, x_3]$, $[x_3, x_5]$, $[x_5, x_7]$, and $[x_7, \infty]$, whose counts are 0, 1, 2, 1, and 0, respectively. During the whole sweeping process, an interval with the maximum count is returned as the final result. The time complexity of this algorithm is $O(n \log n)$, where n is the number of rectangles, since n insertions and n deletions are performed during the sweep, and the cost of each tree operation is $O(\log n)$. This is the best efficiency possible in terms of the number of comparisons [Imai and Asano 1983].

Unfortunately, this algorithm cannot be directly applied to our environment that is focused on massive datasets, since the plane-sweep algorithm is an in-memory algorithm based on the RAM model. Furthermore, a straightforward adaptation of using the B-tree instead of the binary tree still requires a large amount of I/O's, in fact $O(N \log_B N)$. Note that the factor of N is very expensive in the sense that linear cost is only $O(N/B)$ in the EM model.

5. EXACT ALGORITHM FOR MAXIMIZING RANGE SUM

In this section, we propose an external-memory algorithm, namely ***ExactMaxRS***, that exactly solves the MaxRS problem in $O((N/B) \log_{M/B} (N/B))$ I/O's. This is known

[Arge et al. 1993; Imai and Asano 1983] to be the lower bound under the comparison model in external memory.

5.1. Overview

Essentially, our solution is based upon the transformation explained in Section 4. Specifically, to transform the MaxRS problem, for each object $o \in O$, we construct a corresponding rectangle r_o which is centered at the location of o and has a weight $w(o)$. All these rectangles have the same size, which is as specified in the original problem. We use R to denote the set of these rectangles. Also, we define two notions which are needed to define our transformed MaxRS problem later:

Definition 3 (*Location-weight*). Let p be a location in P, the infinite set of points in the entire data space. Its location-weight with regard to R equals the sum of the weights of all the rectangles in R that cover p.

Definition 4 (*Max-region*). The max-region ρ with regard to R is a rectangle such that:

— every point in ρ has the same location-weight τ, and
— no point in the data space has a location-weight higher than τ.

Intuitively, the max-region ρ with regard to R is an intersecting region with the maximum *sum* of the weights of the overlapping rectangles. Then our transformed MaxRS problem can be defined as follows:

Definition 5 (*Transformed MaxRS Problem*). Given R, find a max-region ρ with regard to R.

Apparently, once the above problem is solved, we can return an arbitrary point in ρ as the answer for the original MaxRS problem.

At a high level, the ExactMaxRS algorithm follows the divide-and-conquer strategy, where the entire datset is recursively divided into mutually disjoint subsets, and then the solutions that are locally obtained in the subsets are combined. The overall process of the ExactMaxRS algorithm is as follows:

(1) Recursively divide the whole space vertically into m sub-spaces, called *slab*s and denoted as $\gamma_1, ..., \gamma_m$, each of which contains roughly the same number of rectangles, until the rectangles belonging to each slab can fit in the main memory.
(2) Compute a solution structure for each slab, called *slab-file*, which represents the local solution to the sub-problem with regard to the slab.
(3) Merge m slab-files to compute the slab-file for the union of the m slabs until the only one slab-file remains.

In this process, we need to consider the following: (1) How to divide the space to guarantee the termination of recursion; (2) how to organize slab-files, and what should be included in a slab-file; (3) how to merge the slab-files without loss of any necessary information for finding the final solution.

5.2. The ExactMaxRS Algorithm

Next we address each of the above considerations, and explain in detail our ExactMaxRS algorithm.

5.2.1. Division Phase. Let us start with describing our method for dividing the entire space. Basically, we recursively divide the space vertically into m slabs along the x-dimension until the number of rectangles in a slab can fit in the main memory. Since a rectangle in R can be large, it is unavoidable that a rectangle may need to be split

into a set of smaller disjoint rectangles as the recursion progresses, which is shown in Figure 3. As a naive approach, we could just insert all the split rectangles into

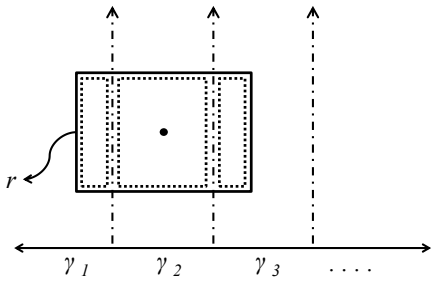


Fig. 3. An example of splitting a rectangle

the corresponding slabs at the next level of recursion. In Figure 3, the three parts of rectangle r will be inserted into slabs γ_1, γ_2, and γ_3, respectively.

However, it is not hard to see that this approach does not guarantee the termination of recursion, since rectangles may *span* an entire slab, e.g., the middle part of r spans slab γ_2. In the extreme case, suppose that all rectangles span a slab γ. Thus, no matter how many times we divide γ into sub-slabs, the number of rectangles in each sub-slab still remains the same, meaning that recursion will never terminate.

Therefore, in order to gradually reduce the number of rectangles for each sub-problem, we do not pass spanning rectangles to the next level of recursion, e.g., the middle part of r will not be inserted in the input of the sub-problem with regard to γ_2. Instead, the spanning rectangles are considered as another local solution for a separate, special, sub-problem. Thus, in the merging phase, the spanning rectangles are also merged along with the other slab-files. In this way, it is guaranteed that recursion will terminate eventually as proved in the following lemma:

LEMMA 1. *After $O(\log_m(N/M))$ recursion steps, the number of rectangles in each slab will fit in the main memory.*

PROOF. Since the spanning rectangles do not flow down to the next recursion step, we can just partition the vertical edges of rectangles. There are initially $2N$ vertical edges. The number of edges in a sub-problem will be reduced by a factor of m by dividing the set of edges into m smaller sets each of which has roughly the same size. Each vertical edge in a slab represents a split rectangle. It is obvious that there exists an h such that $2N/m^h \leq M$. The smallest such h is thus $O(\log_m(N/M))$. □

Determination of m.
We set $m = \Theta(M/B)$, where M/B is the number of blocks in the main memory.

5.2.2. Slab-files. The next important question is how to organize a slab-file. What the question truly asks about is what structure should be returned after *conquering* the sub-problem with regard to a slab. Each slab-file should have enough information to find the final solution after all the merging phases.

To get the intuition behind our solution (to be clarified shortly), let us first consider an easy scenario where every rectangle has weight 1, and is small enough to be totally inside a slab, which is shown in Figure 4. Thus, no spanning rectangle exists. In this case, all we have to do is to just maintain a max-region (black-filled in Figure 4) with regard to rectangles in each slab. Recall that a max-region is the most overlapped area

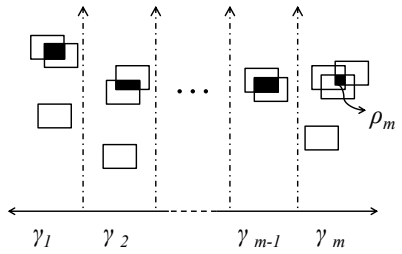


Fig. 4. An easy scenario to illustrate the intuition of slab-files

with respect to the rectangles in the corresponding slab (see Definition 4). Then, in the merging phase, among m max-regions (i.e., one for each slab), we can choose the best one as the final solution. In Figure 4, for instance, the best one is ρ_m because it is the intersection of 3 rectangles, whereas the number is 2 for the max regions of the other slabs.

Extending the above idea, we further observe that the horizontal boundaries of a max-region are laid on the horizontal lines passing the bottom or top edge of a certain rectangle. Let us use the term *h-line* to refer to a horizontal line passing a horizontal edge of an input rectangle. Therefore, for each h-line in a slab, it suffices to maintain a segment that could belong to the max-region of the slab. To formalize this intuition, we define *max-interval* as follows:

Definition 6 (*Max-interval*). Let (1) $\ell.y$ be the y-coordinate of a h-line ℓ, and ℓ_1 and ℓ_2 be the consecutive h-lines such that $\ell_1.y < \ell_2.y$, (2) $\ell \cap \gamma$ be the part of a h-line ℓ in a slab γ, and (3) r_γ be the rectangle formed by $\ell_1.y$, $\ell_2.y$, and vertical boundaries of γ. A max-interval is a segment t on $\ell_1 \cap \gamma$ such that, the x-range of t is the x-range of the rectangle r_{max} bounded by $\ell_1.y$, $\ell_2.y$, and vertical lines at x_i and x_j, where each point in r_{max} has the maximum location-weight among all points in r_γ.

Figure 5 illustrates Definition 6.

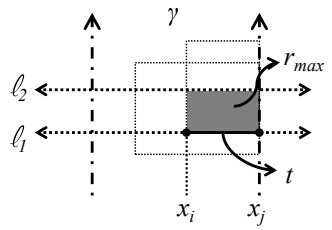


Fig. 5. An illustration of Definition 6

Our slab-file is a set of max-intervals defined *only on h-lines*. Specifically, each max-interval is represented as a tuple specified as follows:

$$t =< y,\ [x_1, x_2],\ sum >$$

where y is the y-coordinate of t (hence, also of the h-line that defines it), and $[x_1, x_2]$ is the x-range of t, and sum is the location-weight of any point in t. In addition, all the tuples in a slab-file should be sorted in ascending order of y-coordinates.

Example 1. Figure 6 shows the slab-files that are generated from the example in Figure 2, assuming that $m = 4$ and $\forall o \in O,\ w(o) = 1$. Max-intervals are represented as solid segments. For instance, the slab-file of slab γ_1 consists of tuples (in this order):

$< y_2,\ [x_1, x_2],\ 1 >$, $< y_4,\ [x_1, x_2],\ 2 >$, $< y_6,\ [x_0, x_2],\ 1 >$, $< y_7,\ [-\infty, x_2],\ 0 >$. The first tuple $< y_2,\ [x_1, x_2],\ 1 >$ implies that, in slab γ_1, on any horizontal line with y-coordinate in (y_2, y_4), the max-interval is always $[x_1, x_2]$, and its sum is 1. Similarly, the second tuple $< y_4,\ [x_1, x_2],\ 2 >$ indicates that, on any horizontal line with y-coordinate in (y_4, y_6), $[x_1, x_2]$ is always the max-interval, and its sum is 2. Note that spanning rectangles have not been counted yet in these slab-files, since (as mentioned earlier) they are not part of the input to the sub-problems with regard to slabs $\gamma_1, ..., \gamma_4$.

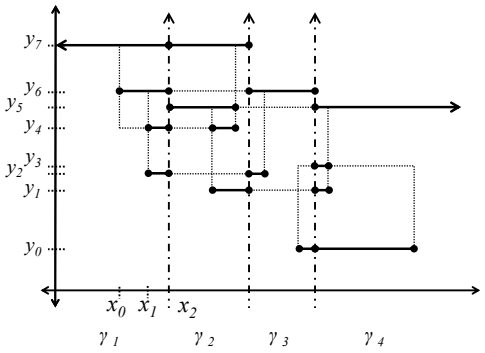


Fig. 6. An example of slab-files

LEMMA 2. *Let K be the number of rectangles in a slab, where $K \leq N$. Then the number of tuples in the corresponding slab-file is $O(K)$.*

PROOF. The number of h-lines is at most double the number of rectangles. As a h-line defines only one max-interval in each slab, the number of tuples in a slab-file is at most $2K$, which is $O(K)$. □

5.2.3. Merging Phase. Now we tackle the last challenge: how to merge the slab-files, which is also the main part of our algorithm.

The merging phase sweeps a horizontal line across the slab-files and the file containing spanning rectangles. At each h-line, we choose a max-interval with the greatest sum among the max-intervals with regard to the m slabs, respectively. Sometimes, max-intervals from adjacent slabs are combined into a longer max-interval.

The details of merging, namely *MergeSweep*, are presented in Algorithm 1. The input includes a set of spanning rectangles and m slab-files. Also, each spanning rectangle contains only the spanning part cropped out of the original rectangle $r_o \in R$, and has the same weight as r_o (recall that the weight of r_o is set to $w(o)$). We use $upSum[i]$ to denote the total weight of spanning rectangles that span slab γ_i and currently intersect the sweeping line; $upSum[i]$ is initially set to 0 (Line 2). Also, we set $t_{slab}[i]$ to be the tuple representing the max-interval of γ_i in the sweeping line. Since we sweep the line from bottom to top, we initially set $t_{slab}[i].y = -\infty$. In addition, the initial interval and sum of $t_{slab}[i]$ are set to be the x-range of γ_i and 0, respectively (Line 3). When the sweeping line encounters the bottom of a spanning rectangle that spans γ_i, we add the weight of the rectangle to $upSum[i]$ (Lines 5 - 6); conversely, when the sweeping line encounters the top of the spanning rectangle, we subtract the weight of the rectangle (Lines 7 - 8). When the sweeping line encounters several tuples (from different slab-files) having the same y-coordinate (Line 9), we first update $t_{slab}[i]$'s accordingly (Lines 10 - 12), and then identify the tuples with the maximum sum among all the $t_{slab}[i]$'s (Line 13). Since there can be multiple tuples with the same maximum sum at an h-line,

we call a function *GetMaxInterval* to generate a single tuple from those tuples (Line 14). Specifically, given a set of tuples with the same sum value, GetMaxInterval simply performs:

(1) If the max-intervals of some of those tuples are consecutive, merge them into one tuple with an extended max-interval.
(2) Return an arbitrary one of the remaining tuples after the above step.

Lastly, we insert the tuple generated from GetMaxInterval into the slab-file to be returned (Line 15). This process will continue until the sweeping line reaches the end of all the slab files and the set of spanning rectangles.

It is worth noting that the MergeSweep algorithm itself requires only $\Theta(m) = \Theta(M/B)$ memory space, since all the in-memory data structures in the algorithm, namely $upSum$, t_{slab}, T, and T', cannot have more than m tuples.

ALGORITHM 1: MergeSweep

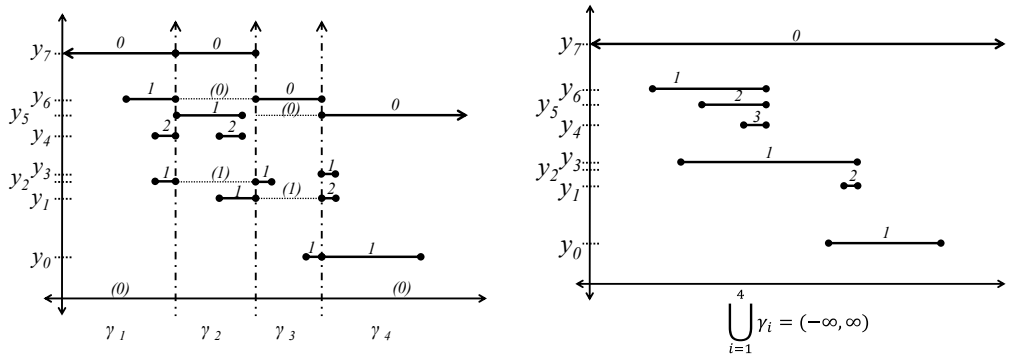
Input: m slab-files $S_1, ..., S_m$ for m slabs $\gamma_1, ..., \gamma_m$, a set of spanning rectangles R'

Output: a slab-file S for slab $\gamma = \bigcup_{i=1}^{m} \gamma_i$. Initially $S \leftarrow \phi$

```
1  for i = 1 to m do
2      upSum[i] ← 0
3      t_slab[i] ← < −∞, the range of x-coordinates of γ_i, 0 >
4  while sweeping the horizontal line ℓ from bottom to top do
5      if ℓ meets the bottom of r_o ∈ R' then
6          upSum[j] ← upSum[j] + w(o), ∀j s.t. r_o spans γ_j
7      if ℓ meets the top of r_o ∈ R' then
8          upSum[j] ← upSum[j] − w(o), ∀j s.t. r_o spans γ_j
9      if ℓ meets a set of tuples T = {t | t.y = ℓ.y} then
10         forall the t ∈ T do
11             t_slab[i] ← t, s.t. t ∈ S_i
12             t_slab[i].sum ← t.sum + upSum[i], s.t. t ∈ S_i
13         T' ← the set of tuples in t_slab[1], ..., t_slab[m] with the largest sum values
14         t_max ← GetMaxInterval(T')
15     S ← S ∪ {t_max}
16 return S
```

Example 2. Figure 7 shows how the MergeSweep algorithm works by using Example 1. For clarity, rectangles are removed, and the sum value of each max-interval is given above the segment representing the max-interval. Also, the value of $upSum$ for each slab is given as a number enclosed in a bracket, e.g., $upSum[2] = 1$, between y_2 and y_6.

When the sweeping line ℓ is located at y_0, two max-intervals from γ_3 and γ_4 are merged into a larger max-interval. On the other hand, when ℓ is located at y_1, the max-interval from γ_4 is chosen, since its sum value 2 is the maximum among the 2 max-intervals at y_1. In addition, it is important to note that sum values of the max-intervals at y_4 and y_5 are increased by the value of $upSum[2] = 1$. Figure 7(b) shows the resulting max-intervals at the end of merging slab-files. We can find that the max-region of the entire data space is between max-intervals at y_4 and y_5, because the max-interval at y_4 has the highest sum value 3.



(a) Four slab-files before merge (b) A slab-file after merge

Fig. 7. An example to illustrate MergeSweep algorithm

We can derive the following lemma:

LEMMA 3. *Let K be the number of rectangles in slab γ in a certain recursion. Given m slab-files $S_1, ..., S_m$ of slabs $\gamma_1, ..., \gamma_m$, s.t., $\gamma = \cup_{i=1}^{m} \gamma_i$, and a set of spanning rectangles R', MergeSweep algorithm returns the slab-file S of γ in $O(K/B)$ I/O's.*

PROOF. Since we set $m = \Theta(M/B)$, a block of memory can be allocated as the input buffer for each slab-file as well as the file containing spanning rectangles. Also, we use another block of memory for the output buffer. By doing this, we can read a tuple of slab-files or a spanning rectangle, or write a tuple to the merged slab-file in $O(1/B)$ I/O's amortized.

The number of I/O's performed by MergeSweep is proportional to the total number of tuples of all slab-files plus the number of spanning rectangles, i.e., $O((|R'| + \sum_{i=1}^{m} |S_i|)/B)$. Let K_i be the number of rectangles in γ_i. Then $|S_i| = O(K_i)$ by Lemma 2. Also, $K_i = \Theta(K/m)$, since the $2K$ vertical edges of the K rectangles are divided into m slabs evenly. Therefore, $\sum_{i=1}^{m} |S_i| = O(K)$, which leads $O((|R'| + \sum_{i=1}^{m} |S_i|)/B) = O(K/B)$, since $|R'| \leq K$. □

5.2.4. Overall Algorithm. The overall recursive algorithm ExactMaxRS is presented in Algorithm 2. We can obtain the final slab-file with regard to a set R of rectangles by calling ExactMaxRS(R, γ, m), where the x-range of γ is $(-\infty, \infty)$. Note that when the input set of rectangles can fit in the main memory, we invoke *PlaneSweep(R)* (Line 8), which is an in-memory algorithm that does not cause any I/O's.

From returned S, we can find the max-region by comparing sum values of tuples trivially. After finding the max-region, an optimal point for the MaxRS problem can be any point in the max-region, as mentioned in Section 5.1.

The correctness of Algorithm 2 is proved by the following lemma and theorem:

LEMMA 4. *Let I^* be a max-interval at a h-line with regard to the entire space and I_1^*,...,I_μ^* be consecutive pieces of I^* for a recursion, each of which belongs to slab γ_i, where $1 \leq i \leq \mu$. Then I_i^* is also the max-interval at the h-line with regard to slab γ_i.*

PROOF. Let $sum(I)$ be the sum value of interval I. To prove the lemma by contradiction, suppose that there exists I_i^* that is not a max-interval in γ_i. Thus, there exists I' in γ_i such that $sum(I') > sum(I_i^*)$ on the same h-line. For any upper level of recursion, if no rectangle spans γ_i, then $sum(I')$ and $sum(I_i^*)$ themselves are already the sum values with regard to the entire space. On the contrary, if there exist rectangles that span γ_i at some upper level of recursion, then the sum values of I' and I_i^* with

ALGORITHM 2: ExactMaxRS

Input: a set of rectangles R, a slab γ, the number of sub-slabs m
Output: a slab-file S for γ

```
if |R| > M then
    Partition γ into γ1,...,γm, which have roughly the same number of rectangles.
    Divide R into R1,...,Rm, R', where Ri is the set of non-spanning rectangles whose left (or
    right) vertical edges are in γi and R' is the set of spanning rectangles.
    for i = 1 to m do
        Si ← ExactMaxRS(Ri, γi, m)
    S ← MergeSweep(S1,...,Sm, R')
else
    S ← PlaneSweep(R)
return S
```

regard to the entire space will be $sum(I') + W_{span}$ and $sum(I_i^*) + W_{span}$, where W_{span} is the total sum of the weights of all the rectangles spanning γ_i in all the upper level of recursion. In both cases above, $sum(I') > sum(I_i^*)$ with regard to the entire space, which contradicts that I^* is the max-interval with regard to the entire space. □

THEOREM 1. *The slab-file returned from the ExactMaxRS algorithm is correct with regard to a given dataset R.*

PROOF. Let ρ^* be the max-region with regard to R, and similarly I^* be the best max-interval that is in fact the bottom edge of ρ^*. Then we want to prove that the algorithm eventually returns a slab-file which contains I^*.

Also, by Lemma 4, we can claim that for any level of recursion, a component interval I_i^* of I^* will also be the max-interval for its h-line within slab γ_i. By Algorithm 1, for each h-line, the best one among the max-intervals at each h-line is selected (perhaps also extended). Therefore, eventually I^* will be selected as a max-interval with regard to the entire space. □

Moreover, we can prove the I/O efficiency of the ExactMaxRS algorithm as in the following theorem:

THEOREM 2. *The ExactMaxRS algorithm solves the MaxRS problem in $O((N/B)\log_{M/B}(N/B))$ I/O's, which is optimal in the EM model among all comparison-based algorithms.*

PROOF. The dataset needs to be sorted by x-coordinates before it is fed into Algorithm 2. The sorting can be done in $O((N/B)\log_{M/B}(N/B))$ I/O's using the textbook-algorithm external sort.

Given a dataset with cardinality N sorted by x-coordinates, the decomposition of the dataset along the x-dimension can be performed in linear time, i.e., $O(N/B)$. Also, by Lemma 3, the total I/O cost of the merging process at each recursion level is also $O(N/B)$, since there can be at most $2N$ rectangles in the input of any recursion. By the proof of Lemma 1, there are $O(\log_{M/B}(N/B))$ levels of recursion. Hence, the total I/O cost is $O((N/B)\log_{M/B}(N/B))$.

The optimality of this I/O complexity follows directly from the results of [Arge et al. 1993] and [Imai and Asano 1983]. □

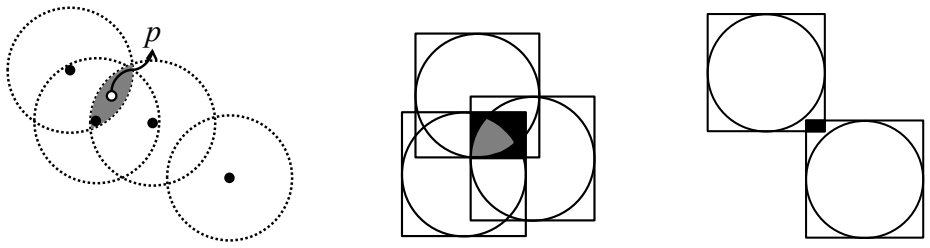
6. APPROXIMATION ALGORITHM FOR MAXIMIZING CIRCULAR RANGE SUM

In this section, we propose an approximation algorithm, namely *ApproxMaxCRS*, for solving the MaxCRS problem (Definition 2). Our algorithm finds an $(1/4)$-approximate solution in $O((N/B)\log_{M/B}(N/B))$ I/O's. We achieve the purpose by a novel reduction that converts the MaxCRS problem to the MaxRS problem.

6.1. The ApproxMaxCRS Algorithm

Recall (from Definition 2) that the goal of the MaxCRS problem is to find a circle with a designated diameter that maximizes the total weight of the objects covered. Denote by d the diameter. Following the idea explained in Section 4, first we transform the MaxCRS problem into the following problem: Let C be a set of circles each of which is centered at a distinct object $o \in O$, has a diameter as specified in the MaxCRS problem, and carries a weight $w(o)$. We want to find a location p in the data space to maximize the total weight of the circles in C covering p. Figure 8(a) shows an instance of the transformed MaxCRS problem, where there are four circles in C, each of which is centered at an object $o \in O$ in the original MaxCRS problem. An optimal answer can be any point in the gray area.

We will use the ExactMaxRS algorithm developed in the previous section as a tool to compute a good approximate answer for the MaxCRS problem. For this purpose, we convert each circle of C to its Minimum Bounding Rectangle (MBR). Obviously, the MBR is a $d \times d$ square. Let R be the set of resulting MBRs. Now, apply ExactMaxRS on R, which outputs the max-region with regard to R. Understandably, the max-region (black area in Figure 8(b)) returned from the ExactMaxRS algorithm may contain locations that are suboptimal for the original MaxCRS problem (in Figure 8(b), only points in the gray area are optimal). Moreover, in the worst case, the max-region may not even intersect with any circle at all as shown in Figure 8(c).



(a) The transformed MaxCRS problem (b) MBRs of circles (c) Worst case

Fig. 8. Converting MaxCRS to MaxRS

Therefore, in order to guarantee the approximation bound, it is insufficient to just return a point in the max region. Instead, our ApproxMaxCRS algorithm returns the best point among the center of the max-region and four *shifted point*s. The algorithm is presented in Algorithm 3.

After obtaining the center point p_0 of the max-region ρ returned from ExactMaxRS function (Lines 2 - 4), we find four shifted points p_i, where $1 \leq i \leq 4$, from p_0 as shown in Figure 9 (Lines 5 - 6). We use σ to denote the *shifting distance* which determines how far a shifted point should be away from the center point. To guarantee the approximation bound as proved in Section 6.2, σ can be set to any value such that $(\sqrt{2}-1)\frac{d}{2} < \sigma < \frac{d}{2}$. Finally, we return the best point $\hat{p}$ among $p_0, ..., p_4$ (Lines 7 - 8).

ALGORITHM 3: ApproxMaxCRS

Input: a set of circles C, the number of slabs m
Output: a point $\hat{p}$
1 Construct a set R of MBRs from C
2 $\gamma \leftarrow$ a slab whose x-range is $(-\infty, \infty)$
3 $\rho \leftarrow$ ExactMaxRS(R, γ, m)
4 $p_0 \leftarrow$ the center point of ρ
5 **for** $i = 1$ *to* 4 **do**
6 $\quad p_i \leftarrow$ GetShiftedPoint(p_0, i)
7 $\hat{p} \leftarrow$ the point p among $p_0, ..., p_4$ that maximizes the total weight of the circles covering p
8 **return** $\hat{p}$

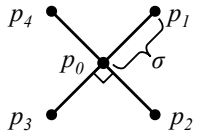


Fig. 9. The illustration of shifting points

Table II. List of symbols in Section 6.2

Symbol	Description
d	the diameter of circles (a given parameter of the MaxCRS problem)
p_0	the centroid of the max-region returned by ExactMaxRS
p_i ($i \in [1, 4]$)	a shifted point described in Algorithm 3
c_i ($i \in [0, 4]$)	the circle with diameter d centering at point p_i
r_0	the MBR of c_0
$O(s)$	the set of objects covered by s, where s is a circle or an MBR
$W(s)$	the total weight of the objects in $O(s)$

Note that Algorithm 3 does not change the I/O complexity of the ExactMaxRS algorithm, since only linear I/O's are required in the entire process other than running the ExactMaxRS algorithm. Note that Line 6 of Algorithm 3 requires only a single scan of C.

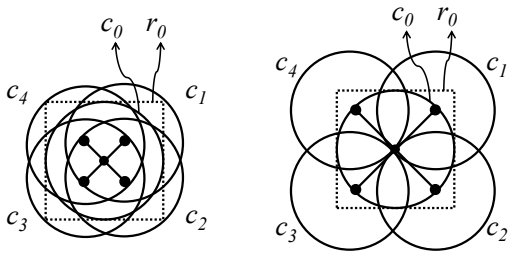
6.2. Approximation Bound

Now, we prove that the ApproxMaxCRS algorithm returns a $(1/4)$-approximate answer to the optimal solution, and also prove that this approximation ratio is tight with regard to this algorithm. To prove the approximation bound, we use the fact that a point p covered by the set of circles (or MBRs) in the transformed MaxCRS problem is truly the point such that the circle (or MBR) centered at p covers the corresponding set of objects in the original MaxCRS problem. The main symbols used only in this section are summarized in Table II.

LEMMA 5. *For each $i \in [0, 4]$, let c_i be the circle centered at point p_i, r_0 be the MBR of c_0, and $O(s)$ be the set of objects covered by s, where s is a circle or an MBR. Then $O(r_0) \subseteq O(c_1) \cup O(c_2) \cup O(c_3) \cup O(c_4)$.*

PROOF. As shown in Figure 10, all the objects covered by r_0 are also covered by c_1, c_2, c_3, or c_4, since $(\sqrt{2} - 1)\frac{d}{2} < \sigma < \frac{d}{2}$. □

Let $W(s)$ be the total weight of the objects covered by s, where s is a circle or an MBR. Then, we have:



(a) $\sigma = (\sqrt{2}-1)\frac{d}{2}$ (b) $\sigma = \frac{d}{2}$

Fig. 10. Lemma 5

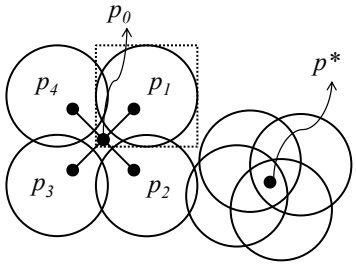


Fig. 11. Theorem 4

LEMMA 6. $W(r_0) \leq 4 \max_{0 \leq i \leq 4} W(c_i)$.

PROOF.

$$\begin{aligned} W(r_0) &\leq \sum_{1 \leq i \leq 4} W(c_i) \text{ (by Lemma 5)} \\ &\leq 4 \max_{0 \leq i \leq 4} W(c_i) \end{aligned}$$

□

THEOREM 3. *The ApproxMaxCRS algorithm returns a* $(1/4)$*-approximate answer to the MaxCRS problem.*

PROOF. Recall that $\hat{p}$ is the point returned from Algorithm 3 as the approximate answer to the MaxCRS problem. Let point p^* be an optimal answer for the MaxCRS problem. Denote by $\hat{r}$ and r^* the MBRs centered at point $\hat{p}$ and p^*, respectively. Likewise, denote by $\hat{c}$ and c^* be the circles centered at point $\hat{p}$ and p^*, respectively. The goal is to prove $W(c^*) \leq 4W(\hat{c})$.

We achieve this purpose with the following derivation:

$$W(c^*) \leq W(r^*) \leq W(r_0) \leq 4 \max_{0 \leq i \leq 4} W(c_i) = 4W(\hat{c})$$

The first inequality is because r^* is the MBR of c^*. The second inequality is because p_0 is the optimal solution for the MaxRS problem on R. The last equality is because ApproxMaxCRS returns the best point among $p_0, ..., p_4$. □

THEOREM 4. *The* $1/4$ *approximation ratio is tight for the ApproxMaxCRS algorithm.*

PROOF. We prove this by giving a worst case example. Consider an instance of the transformed MaxCRS problem in Figure 11 where each circle has weight 1. In this case, we may end up finding a max-region centered at p_0 using the ExactMaxRS algorithm. This is because both p_0 and p^* are covered by 4 MBRs, and MergeSweep (see Algorithm 1) chooses an arbitrary one in such a *tie* case. Supposed that p_0 is the center of the max-region returned from ExactMaxRS, we will choose one of $p_1, ..., p_4$ as an approximate solution. Since each of $p_1, ..., p_4$ is covered by only 1 circle, our answer is $(1/4)$-approximate, because the optimal answer p^* is covered by 4 circles. □

7. THEORETICAL STUDY ON THE PROBLEM OF FINDING ALL THE MAX-REGIONS

In this section, we discuss an extended version of the MaxRS problem, namely *All-MaxRS*, where all the max-regions are retrieved while only one max-region is returned

Table III. List of symbols in Section 7.1

Symbol	Description
k	the count of rectangles covering each max-region w.r.t. R
ρ_k	a max-region covered by k rectangles
$\mathcal{M}_k$	the set of all ρ_k's
$\overline{\lvert\mathcal{M}_k\rvert}$	the upper bound of the number of all ρ_k's

in Section 5. This is a natural extension in the sense that there can be many max-regions whose location-weights are equal to the maximum especially when a dataset is dense. By slightly modifying Definition 5, the AllMaxRS problem can be defined as follows:

Definition 7 (*AllMaxRS Problem*). Given a set R of N rectangles with the same size, find all the max-regions w.r.t. R.

Surprisingly, it is not trivial at all to extend the ExactMaxRS algorithm for solving the AllMaxRS problem. The straightforward extension of ExactMaxRS can be very inefficient especially when there are lots of tied max-intervals. In Section 7.1, we first examine the upper bound of the number of tied max-regions, and thereby prove that a simple extension of ExactMaxRS is significantly inefficient in terms of the I/O complexity in Section 7.2.

7.1. The Upper Bound of the Number of Max-regions

In this section, we examine the upper bound of the number of max-regions; at most how many tied max-regions w.r.t. R can exist in the extreme case. For the clarity of our argument, we first assume that the weight of every rectangle is equally set to 1 (i.e., unweighted), and then generalize the argument to the case where the weights of rectangles can be different.

In the unweighted version of the AllMaxRS problem, max-regions are just the most intersecting regions. Let us use the term *count*, instead of the location-weight, to refer to the number of intersecting rectangles. We use k to denote the count of each max-region w.r.t. R, and use ρ_k to denote such a max-region covered by k rectangles. Also, we use $\mathcal{M}_k$ to denote the set of all ρ_k's, which should be returned by the AllMaxRS problem. Then our goal is to find the upper bound of $|\mathcal{M}_k|$, denoted by $\overline{|\mathcal{M}_k|}$. New symbols frequently used in this section are presented in Table III.

We first focus on figuring out how many max-regions can exist inside a rectangle as follows:

LEMMA 7. *There exist at most k^2 max-regions inside a rectangle $r \in R$.*

PROOF. See appendix. □

Lemma 7 derives the following important corollary:

COROLLARY 1. $\overline{|\mathcal{M}_k|} < \frac{k^2 N}{k} = kN$

PROOF. This is proved by the fact that there are at most k^2N max-regions for N rectangles by Lemma 7 and each max-region is covered by k different rectangles. Since not every rectangle can have k^2 max-regions inside it, the total number of max-regions cannot exceed k^2N/k. □

We now examine the upper bound of $|\mathcal{M}_k|$ which is tight in the worst case as the following theorem:

THEOREM 5. *Given a set R of N unweighted rectangles with the same size, there can exist at most $\Theta(kN)$ max-regions in the arrangement of R, where k is the count of each max-region.*

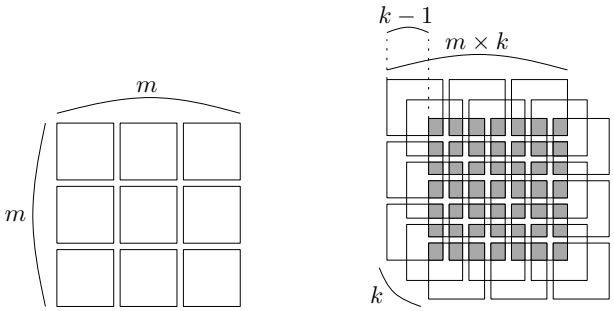
PROOF. Let $N = m^2k$ for some $m \geq 2$ without loss of generality. This can be seen that R consists of k *plates* of $m \times m$ rectangles (see Figure 12(a)). By overlapping k plates as shown in Figure 12(b), we have $(mk-(k-1))^2$ max-regions even if this may not be the arrangement of R that maximizes $|\mathcal{M}_k|$. Thus, we have:

$$\begin{aligned}
\overline{|\mathcal{M}_k|} \;&\geq\; |\mathcal{M}_k| = (mk-(k-1))^2 = m^2k^2 - 2mk(k-1) + (k-1)^2 \\
&=\; kN - 2(k-1)\sqrt{kN} + (k-1)^2 \;\; \text{(by } N = m^2k\text{)} \\
&=\; \frac{N^2}{f} - \frac{2N^2}{f}\sqrt{\frac{1}{f}} + 2N\sqrt{\frac{1}{f}} + \frac{N^2}{f^2} - \frac{2N}{f} + 1 \;\; \text{(by } k \leftarrow \tfrac{N}{f}\text{)} \\
&=\; \frac{N^2}{f}(1 - 2\sqrt{\frac{1}{f}} + \frac{1}{f}) + 2N\sqrt{\frac{1}{f}} - \frac{2N}{f} + 1 \\
&>\; \frac{N^2}{4f} + 2N\sqrt{\frac{1}{f}} - \frac{2N}{f} \;\; \text{(by } f = m^2 \geq 4\text{)} \\
&=\; \frac{kN}{4} + 2\sqrt{kN} - 2k. \qquad (1)
\end{aligned}$$

Finally, by considering Corollary 1 together with (1), for all $k \leq \frac{N}{4}$, there exists $c > 0$ such that:

$$ckN \leq \overline{|\mathcal{M}_k|} < kN.$$

□



(a) A *plate* of $m \times m$ rectangles (b) k overlapping *plate*s

Fig. 12. The illustration of Theorem 5

Theorem 5 is based upon the assumption that the weight of every rectangle is equally set to 1. By the following theorem, we claim that the same argument is also valid with respect to a set of weighted rectangles:

THEOREM 6. *Given a set R of N weighted rectangles with the same size, let W_{max} denote the location-weight of each max-region, and $\overline{k}$ denote the maximum number of intersecting rectangles whose total weight equals W_{max}. Then the number of max-regions w.r.t. R is less than $\overline{k}N$.*

PROOF. See appendix. □

7.2. A Simple Extension of ExactMaxRS

By slightly modifying the ExactMaxRS algorithm that solves the MaxRS problem, we can also solve the AllMaxRS problem. To retrieve all the max-regions w.r.t. R, each slab-file should maintain all the tied max-intervals, instead of one max-interval, on every h-line. Thus, each tuple of a slab-file has the following form:

$$t =< y,\ \{[x_1, x_2], [x_3, x_4], ..., [x_i, x_j]\},\ sum >$$

where $\{[x_1, x_2], [x_3, x_4], ..., [x_i, x_j]\}$ is the set of separated max-intervals whose location-weights are equal to sum. In the merging phase of ExactMaxRS, for each h-line, we generate this set of separated max-intervals from multiple tuples with the maximum sum among all the tuples on the h-line. By doing this, it is guaranteed that every max-region w.r.t. R must be retrieved, since all the max-intervals are obviously contained in the final slab-file. We call this simple extension of ExactMaxRS *SimpleAllMaxRS*.

However, the SimpleAllMaxRS algorithm is significantly inefficient in the worst case where there are many tied max-intervals on a same h-line in every recursion. The worst case I/O complexity is proved as the following theorem:

THEOREM 7. *SimpleAllMaxRS solves the AllMaxRS problem in* $O((kN^2/B)\log_{M/B}(N/B))$ *I/O's in the worst case, where* k *is the maximum number of rectangles containing a max-region w.r.t.* R.

PROOF. By Theorems 5 and 6, if there are K rectangles in γ, there can be at most $\Theta(kK)$ max-regions in γ. Even though some of the rectangles in γ can be cropped by the boundaries of γ, they do not change the upper bound of the number of max-regions in the worst case.

Furthermore, to determine the top of each max-region, we should maintain all the max-intervals on every h-line that intersects with the max-region. Thus, every max-interval should be duplicated in many times, which is in fact at most $O(N)$ times in the worst case.

Therefore, the number of tuples in the slab-file of γ is at most $\Theta(kK^2)$ in the worst case. By the proofs of Lemmas 2 and 3, we can claim that the MergeSweep algorithm requires $O(kK^2/B)$ I/O's to create a slab-file of γ. Therefore, the total I/O cost of the worst case is $O((kN^2/B)\log_{M/B}(N/B))$, which follows from the proof of Theorem 2. □

Besides its high I/O complexity, another crucial disadvantage of SimpleAllMaxRS lies in the fact that it is not an output-sensitive algorithm. Thus, regardless of the number of max-regions to be returned, SimpleAllMaxRS maintains all the max-intervals in every slab file, even if those *local* max-intervals are not truly the *global* max-intervals w.r.t. R. This limitation is due to the fact that, in the basic processing scheme of ExactMaxRS, we cannot be sure which max-intervals are also the global max-intervals until the final merging process is finished.

8. TWO-PHASE OUTPUT-SENSITIVE ALGORITHM FOR FINDING ALL THE MAX-REGIONS

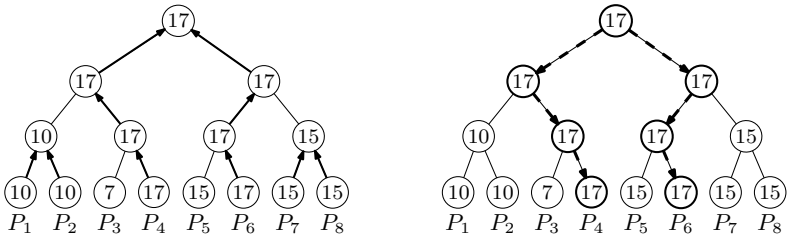
Motivated by the limitation of SimpleAllMaxRS, we devise a two-phase output-sensitive algorithm, called *TwoPhaseMaxRS*, for solving the AllMaxRS problem. TwoPhaseMaxRS exactly retrieves all the max-regions in $O(((N+T)/B)\log_{M/B}(N/B))$ I/O's, where T is the number of max-regions to be returned.

8.1. Overview

We first outline the intuition behind TwoPhaseMaxRS. To this end, let us imagine a tournament where multiple *co-winner*s are allowed, which is a *metaphor* for the process of SimpleAllMaxRS. In this tournament, to be the final winners (max-intervals w.r.t. R), each player (each max-interval w.r.t. a slab) should beat every competitor in

every single match (in the merging phase of every recursion). In this perspective, an important observation is that SimpleAllMaxRS just sends all the winners up to the next level until the final winners are decided. This approach is natural, but could be inefficient in the sense that not all of the winners can be the final winners.

Our remedy for this problem is to keep track of the *path* information of all the final winners, and thereby considering only the winners who will be selected as the final winners. To this end, we adopt a two-phase approach. In the first phase, we record only the score of the winners for each match in a bottom-up manner, as shown in Figure 13(a). After that, in the second phase, we downwardly *mark* the players only who have the maximum score as recorded in the first phase, as shown in Figure 13(b). Finally, we return only the winners who are marked (i.e., P_4 and P_6 in Figure 13).



(a) The first phase (b) The second phase

Fig. 13. The illustration of marking winners in a tournament

Now we apply the above intuition to the AllMaxRS problem. To build the path information of all the max-regions, rather than maintaining all the max-intervals per h-line, we store h-lines themselves augmented with their maximum location-weights in slab-files. For each h-line in slab-files, we *mark* the h-line, only if it actually contains any global max-intervals (i.e., final winners). After that, by considering only marked h-lines in all these slab-files, we can finally construct all the max-regions w.r.t. R efficiently.

The overall process of the TwoPhaseMaxRS algorithm is as follows:
In the first phase,

(1) During the top-down process, recursively divide the data space (and the set of rectangles) vertically into m slabs (and m subsets of rectangles), as if doing in ExactMaxRS and SimpleAllMaxRS.
(2) During the bottom-up process, construct the slab-file for each slab by merging the slab-files in the lower level of recursion.

In the second phase,

(1) During the top-down process, recursively mark all the h-lines in each slab-file that could constitute the path information of all the global max-intervals.
(2) During the bottom-up process, retrieve all the max-regions w.r.t. R by gradually constructing max-regions only laid on marked h-lines.

In this process, we will encounter the following questions: (1) How to reorganize the structure of the slab-file in such a way that the path information of all the max-regions will be stored; (2) how to merge such reorganized slab-files; (3) how the marking process is performed to complete the path information of all the max-regions; (4) how to finally construct the max-regions using the path information built in slab-files.

While addressing all these questions, the most important and challenging, in particular theoretically, mission is that the entire process must be performed in an *output-sensitive* manner.

8.2. The TwoPhaseMaxRS Algorithm

We now address each of the above questions, and describe the details of the TwoPhaseMaxRS algorithm.

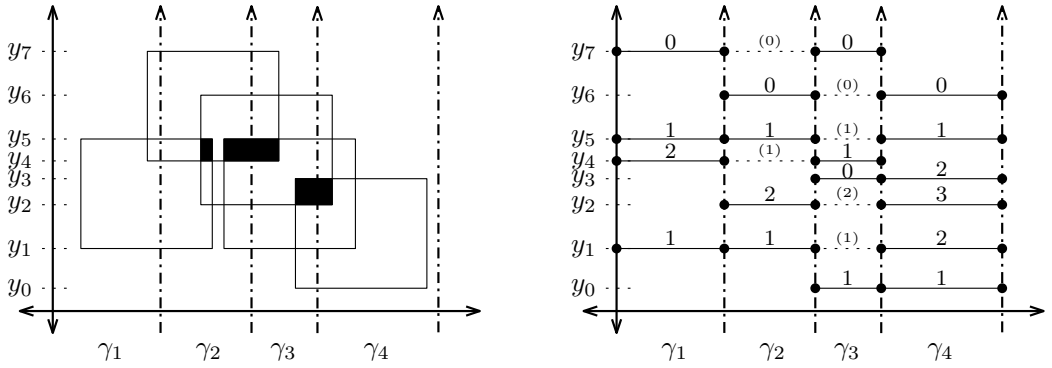
8.2.1. Slab-files Revisited. For the TwoPhaseMaxRS algorithm, we use slab-files for a different purpose from those of ExactMaxRS and SimpleAllMaxRS. Instead of directly obtaining the solution (i.e., max-region) from slab-files, we store (acquire) the path information in (from) slab-files. Therefore, each slab-file needs to record no longer max-intervals, but just h-lines attached with location-weights and other additional information.

Now each tuple t of slab-files has the following form to represent an h-line:

$$t = < y,\ sum,\ marked > .$$

y and sum are defined as before, but note that the x-range $[x_1, x_2]$ is removed. Thus, from t, we cannot know exact max-intervals at y, but only the fact that there exist one or more max-intervals at y and their location-weights are equal to sum. In order to mark h-lines, we use $marked$ that can be either $true$ or $false$. When we construct max-regions from the lowest level of recursion, only marked (i.e., $marked = true$) h-lines are considered.

Example 3. Figure 14(a) introduces a running example to illustrate the TwoPhaseMaxRS algorithm. Assuming that every rectangle's weight is 1, there are three max-regions (black-filled regions) each of which is covered by three rectangles. Given $m = 4$ (i.e., four slabs), Figure 14(b) shows the corresponding slab-files in the modified form. Since we do not consider the x-ranges of max-intervals, each tuple (i.e., h-line) is represented as a line segment spanning its slab.



(a) Multiple max-regions w.r.t R (b) Slab-files (revisited)

Fig. 14. A running example to illustrate TwoPhaseMaxRS

8.2.2. First Phase. We now explain the details of the first phase. The top-down dividing scheme of TwoPhaseMaxRS is exactly the same as ExactMaxRS or SimpleAllMaxRS; we therefore describe only the revised merging process, called *PathSweep*, which is performed during the bottom-up process.

The goal of PathSweep is basically to generate a merged slab-file for the upper level of recursion, and hence its basic flow, as represented in Algorithm 4, is very similar to MergeSweep. One difference is that in PathSweep we do not need a function to generate a (extended) single interval from tied several max-intervals (recall a function "GetMaxInterval" in MergeSweep). Instead, we generate a tuple with the maximum

sum, which represents just the h-line where the current sweeping line is located (Lines 12 - 13). It is worth noting that $t_{slab}[i]$ indicates a tuple in S_i having the maximum sum at the current sweeping line, and it can be below the tuple in the upper level slab-file that it actually contributes to. Thus, a tuple in a slab-file does not always contribute to the tuple in the same line in the upper level slab-file. Finally, if there has been a change about the current max-intervals, the newly generated tuple is inserted into the slab-file to be returned (Lines 14 - 15). Note that we can always know the max-intervals at the current sweeping line during the entire sweeping process, even though such max-intervals are not stored in slab-files.

ALGORITHM 4: PathSweep

Input: m slab-files $S_1, ..., S_m$ for m slabs $\gamma_1, ..., \gamma_m$, a set of spanning rectangles R'

Output: a slab-file S for slab $\gamma = \bigcup_{i=1}^{m} \gamma_i$. Initially $S \leftarrow \phi$

1 **for** $i = 1$ *to* m **do**
2 $\quad upSum[i] \leftarrow 0$
3 $\quad t_{slab}[i] \leftarrow < -\infty,\ 0,\ false >$
4 **while** *sweeping the horizontal line* ℓ *from bottom to top* **do**
5 $\quad$ **if** ℓ *meets the bottom of* $r_o \in R'$ **then**
6 $\quad\quad upSum[j] \leftarrow upSum[j] + w(o)$, $\forall j$ s.t. r_o spans γ_j
7 $\quad$ **if** ℓ *meets the top of* $r_o \in R'$ **then**
8 $\quad\quad upSum[j] \leftarrow upSum[j] - w(o)$, $\forall j$ s.t. r_o spans γ_j
9 $\quad$ **if** ℓ *meets a set of tuples* $T = \{t \mid t.y = \ell.y\}$ **then**
10 $\quad\quad$ **foreach** $t \in T$ **do**
11 $\quad\quad\quad t_{slab}[i] \leftarrow t$, s.t. $t \in S_i$
12 $\quad$ **if** ℓ *meets any tuples or edges of spanning rectangles* **then**
13 $\quad\quad t_{max} \leftarrow < \ell.y,\ \max_{i \in [0,m]}\{t_{slab}[i].sum + upSum[i]\},\ false >$
14 $\quad\quad$ **if** *there has been a change from the previous* $t_{max}.sum$ *or the set of max-intervals that contributed to the previous* t_{max} **then**
15 $\quad\quad\quad S \leftarrow S \cup \{t_{max}\}$
16 **return** S

Example 4. Figure 15 illustrates the result of PathSweep by using Example 3, which is the resulting slab-file of the entire space. From this final slab-file, we know that one or more max-regions are laid on the h-lines at y_2 and y_4, and the global maximum sum is 3.

LEMMA 8. *Let K be the number of rectangles in slab γ in a certain recursion. Given m slab-files and a set of spanning rectangles, the PathSweep algorithm returns the slab-file of γ in $O(K/B)$ I/O's.*

PROOF. Every tuple in a slab-file complying to the modified form corresponds to a unique h-line, and the number of h-lines in γ is $O(K)$ by Lemma 2. Similar to the proof of Lemma 3, the I/O cost is proportional to the total number of tuples in files, which implies that the total I/O cost is $O(K/B)$. □

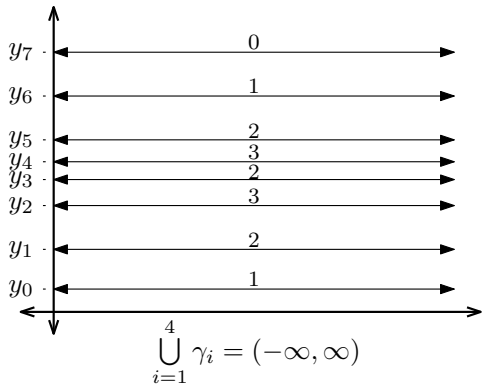


Fig. 15. A merged slab-file for the entire space

8.2.3. Second Phase. By using the slab-files that are created in the first phase, in the second phase we actually construct the max-regions. During the top-down process, we recursively mark every h-line whose maximum location-weight (i.e., sum) is the same as that of the h-line already marked in the upper level of recursion. Through this, we can build the path information of the max-regions w.r.t. R. After that, during the bottom-up process, we retrieve all the max-regions by sending up only max-regions that are laid on the marked h-lines in slab-files.

Top-down process.
Let us first examine how we can mark the h-lines that constitute the path information of the final max-regions. Marking h-lines is basically the process of finding out where (which slabs) each max-region comes from over the entire recursions. This marking process, called *MarkSweep*, can be done by sweeping a horizontal line downwardly across the slab-files together with their merged slab-file and the file of spanning rectangles.

The details of MarkSweep are shown in Algorithm 5. One merged slab-file S, m slab-files $S_1, ..., S_m$, and the file R' of spanning rectangles are given as the input. As mentioned in Section 8.2.1, each tuple t of slab-files indicates that there exist one or more max-intervals with $t.sum$ at $t.y$. Therefore, as an initial step, in the case of final slab-file for the entire space, we mark every h-line (i.e., tuple) whose sum equals the global maximum sum, denoted by W_{max} (Lines 1 - 4).

Next, we need to determine where all the marked h-lines in S originate from. To this end, let us first define a notion "*promote*" as follows:

Definition 8 (*Promotion*). Let S be a slab-file of γ, and S_i be a slab-file of $\gamma_i \subset \gamma$. Then we say that $t_i \in S_i$ is *promoted* to $t \in S$ when t_i is selected in the merging process as a tuple with the maximum sum (plus the total weight of spanning rectangles) at $t.y$.

Intuitively, "*a tuple is promoted*" means that the tuple is selected as a *winner* in the merging step. For example, in Figure 14(b), the tuple at y_1 in γ_4 is promoted to the tuple at y_1 in Figure 15, but tuples at y_1 in γ_1 or γ_2 are not.

It is worth noting that a tuple, say $t_i \in S_i$, can be below its promoted tuple $t \in S$ that t_i actually contributes to. For example, the tuple at y_6 in Figure 15 actually originates from the tuple at y_5 in γ_1 in Figure 14(b).

This brings us a question how we can determine whether a tuple in S has been promoted by a tuple in S_i. This can be answered by the following lemma:

LEMMA 9. *Let S be a slab-file of γ, and S_i be a slab-file of $\gamma_i \subset \gamma$. Then $t \in S$ is promoted by $t_i \in S_i$ iff. (1) t_i is in the same line as t or immediately below t, and (2) $t_i.sum$ equals $t.sum - W_{span}[i]$, where $W_{span}[i]$ is the total weight of rectangles spanning γ_i at $t.y$.*

PROOF. (1) implies that there is no tuple in S_i at any horizontal line between $t_i.y$ and $t.y$. Therefore, before S_i is merged, in γ_i, the maximum location-weight at $t.y$ is $t_i.sum$ by the definition of the tuple in slab-files. When we merge all S_i's along with the set R' of spanning rectangles (by PathSweep), we increase the location-weight accordingly by the total weight of spanning rectangles. Thus, we must also increase the location-weight by $W_{span}[i]$ when the sweeping line is at $t.y$, and hence the maximum location-weight at $t.y$ in γ_i becomes $t_i.sum + W_{span}[i]$. Since, in γ, the maximum location-weight at $t.y$ is specified as $t.sum$ by S and $t.sum = t_i.sum + W_{span}[i]$ by (2), t_i must be selected for (promoted to) a tuple at $t.y$ in S (i.e., $t \in S$). □

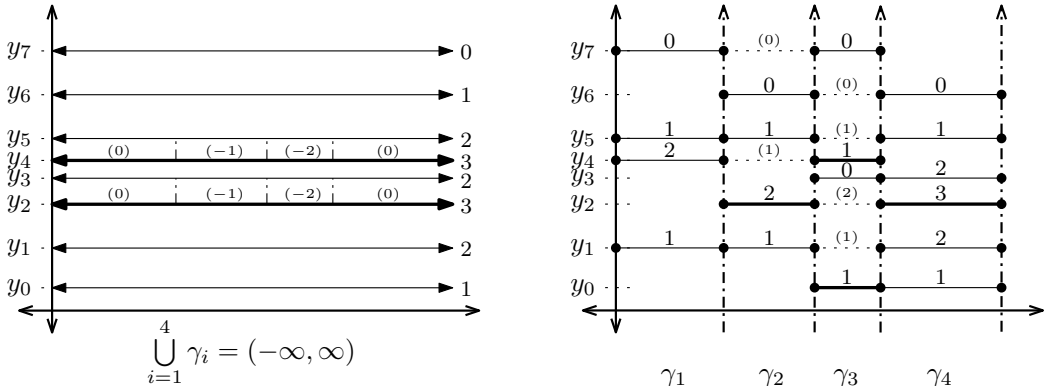
Our next step is to find and mark all the tuples (i.e., h-lines) in S_i's that have been promoted to the tuples already marked in S. We use $maxSum[i]$ to maintain $t.sum - W_{span}[i]$, as mentioned in Lemma 9, for each marked $t \in S$. $upSum[i]$ is also used as before. When the sweeping line encounters a marked tuple $t \in S$, we update $maxSum[i]$ to be compared with sum values of tuples in S_i's (Lines 8 - 11). When the sweeping line encounters a tuple $t_i \in S_i$, we mark t_i only if t_i can be promoted to the current marked tuple $t_{max} \in S$ based on Lemma 9 (Lines 13 - 14). When the sweeping line encounters the top (bottom) of a spanning rectangle that spans γ_j, we subtract (add) the weight of the spanning rectangle from (to) $upSum[j]$ (Lines 15 - 18). Note that the way of tackling the weight of each spanning rectangle is exactly the same as PathSweep even though the sweeping direction here is the opposite. This is why we can just add current $upSum[i]$ in order to calculate $maxSum[i]$ (Line 11).

ALGORITHM 5: MarkSweep

Input: a slab-file S of γ, m slab-files $S_1, ..., S_m$ for m sub-slabs of γ, a set of spanning rectangles R'

```
1  if γ is the entire space then
2      foreach t ∈ S do
3          if t.sum = W_max then
4              t.marked ← true
5  for i = 1 to m do
6      maxSum[i] ← 0, upSum[i] ← 0
7  while sweeping the horizontal line ℓ from top to bottom do
8      if ℓ meets t ∈ S, s.t. t.marked = true then
9          t_max ← t
10         for i = 1 to m do
11             maxSum[i] ← t_max.sum + upSum[i]
12     if ℓ meets t_i ∈ S_i then
13         if (maxSum[i] = t_i.sum) ∧ (t_i is immediately below or in the same line as t_max) then
14             t_i.marked ← true
15     if ℓ meets the bottom of r_o ∈ R' then
16         upSum[j] ← upSum[j] + w(o), ∀j s.t. r_o spans γ_j
17     if ℓ meets the top of r_o ∈ R' then
18         upSum[j] ← upSum[j] − w(o), ∀j s.t. r_o spans γ_j
```

Example 5. Figure 16 shows the result of our example after MarkSweep. In Figure 16(a), only h-lines with $sum = 3$ are marked (represented as bold lines) in the final slab-file, due to $W_{max} = 3$. Also, $upSum[i]$'s at each marked h-line are given as numbers in parenthesis. Figure 16(b) shows four slab-files where h-lines to be promoted in the final slab-file are marked. For example, a tuple at y_0 in γ_3 is marked because it is immediately below the marked tuple at y_2 and $W_{max} + upSum[i] = 3 + (-2) = 1$ equals its $sum = 1$.



(a) The final slab-file after MarkSweep (b) Four slab-files after MarkSweep

Fig. 16. The example to illustrate the MarkSweep algorithm

LEMMA 10. *Let K be the number of rectangles in slab γ in a certain recursion. Given a slab-file S of γ, m slab-files $S_1, ..., S_m$ of slabs $\gamma_1, ..., \gamma_m$, s.t., $\gamma = \cup_{i=1}^{m} \gamma_i$, and a set R' of spanning rectangles, the MarkSweep algorithm is performed in $O(K/B)$ I/O's.*

PROOF. Let us first address that the sweeping direction of MarkSweep is exactly the opposite to that of PathSweep, but every slab-file is sorted in ascending order of y-coordinates. Fortunately, this can be easily done without any overhead in terms of I/O cost. Consider tuples residing in two consecutive blocks B_{i-1} and B_i in a slab-file such that B_i is next to B_{i-1} in ascending order. Thus, all tuples in B_i are not below any tuples in B_{i-1}. To access tuples downwardly, we can just load B_{i-1}, instead of B_{i+1}, followed by B_i, and sort tuples loaded in memory in descending order of y-coordinates. This does not cause any additional I/O's.

In common with PathSweep and MergeSweep, the total I/O cost is proportional to the total number of tuples in files, which is $O(|S|/B + |S_1|/B + ... + |S_m|/B + |R'|/B) = O(K/B)$ □

Region-files.

Now we have all the path information of final max-regions, which are built in slab-files. To actually store and return max-regions, we need another type of data structures for each slab, called *region-files*. A region-file of slab γ is basically the set of rectangles that are necessary for constructing the final max-regions residing in γ. More specifically, each rectangle in a region-file can be regarded as an intermediate result of a final max-region, which is in fact identical to r_{max} in Definition 6. We use the term *max-range* to distinguish this intermediate rectangle from its final max-region. Each max-range in a region-file is formally defined as follows:

Definition 9 (*Max-range*). Let t be a marked tuple in a slab-file of slab γ. Then a max-range u is a rectangle in γ such that (1) the bottom-edge of u is on t, (2) the x-range of u is the x-range of a certain max-interval on t, denoted by $[x_1, x_2]$, and (3) the top-edge of u is on the lowest h-line such that the location-weight of $[x_1, x_2]$ at the h-line is not equal to $t.sum$.

It is important to note that a max-range is not necessarily fully-contained between two consecutive h-lines, unlike r_{max} in Definition 6. This is because, in PathSweep, we insert a new tuple (i.e., h-line) even when a new max-interval with the same location-weight is added, which is different from that a tuple (i.e., max-interval) is inserted only if the maximum location-weight is changed in MergeSweep. Each max-range needs both the x-range and the y-range as specified by the following tuple u:

$$u =< [y_1, y_2],\ [x_1, x_2] > .$$

Example 6. Figure 17 shows four region-files that are created from our example in Figure 14(a). Note that spanning rectangles are not counted here. Each max-range is represented as a gray-filled rectangle attached with its sum value. Thus, the region-file of γ_1 is empty. Other rectangles represent the intermediate results generated when we initially create the region-files at the lowest level (i.e., in memory). Among these intermediate results, we store and send up only ones whose bottom edges are on the marked h-lines, which are the max-ranges.

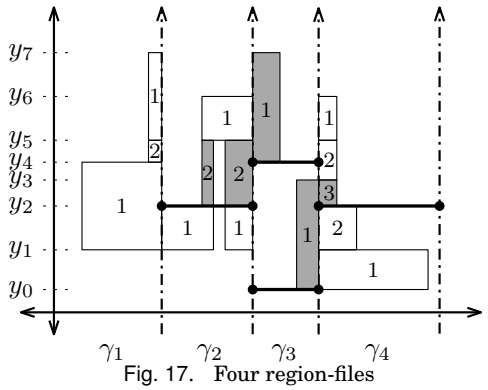


Fig. 17. Four region-files

LEMMA 11. *Let T_i be the number of max-regions w.r.t. R residing in slab γ_i, i.e., their side edges are in slab γ_i. Then the number of max-ranges in the corresponding region-file X_i is $O(T_i)$.*

PROOF. We prove this lemma by showing that each max-range in X_i contains at least one max-region w.r.t. R. Let u_i be a max-range in X_i. By Definition 9, the bottom edge of u_i is on a marked h-line $t_i \in S_i$, where S_i is the slab-file corresponding to X_i. Since t_i is marked, we can guarantee that t_i is eventually promoted to a marked h-line $t_{max} \in S$, where S is the final slab-file w.r.t. the entire space. Thus, after all the merging steps, the maximum location-weight at $t_{max}.y$ in γ_i, which could be either in the same line as t_i or above t_i, will be $t_i.sum + W_{span} = t_{max}.sum = W_{max}$, where W_{span} is the total weight of all the spanning rectangles in γ_i at $t_{max}.y$.

Now we show that t_{max} must intersect u_i (i.e., $u_i.y_1 \leq t_{max}.y < u_i.y_2$). This implies that there exists a region in u_i whose location-weight is W_{max} after all the merging steps, which completes the proof of this lemma. It is obvious to prove $u_i.y_1 \leq t_{max}.y$, as $u_i.y_1 = t_i.y \leq t_{max}.y$.

To prove $t_{max}.y < u_i.y_2$ by contradiction, suppose $u_i.y_2 \leq t_{max}.y$. Let $t'_i \in S_i$ be another h-line that contains the top edge of u_i. Then we have $t_i.y < t'_i.y \leq t_{max}.y$.

Such t'_i must exist in S_i, because the x-range of u_i, i.e., $[u_i.x_1, u_i.x_2]$, will be removed from the set of max-intervals above t'_i by Definition 9 (recall that a new tuple is inserted by PathSweep whenever there has been any changes in the max-intervals). Moreover, $[u_i.x_1, u_i.x_2]$ is always the max-interval w.r.t. any upper level slabs. This is because t_i is a marked tuple, which means $[u_i.x_1, u_i.x_2]$ can never be discarded by other max-intervals during the entire merging step. Since $[u_i.x_1, u_i.x_2]$ is always removed from the set of max-intervals above $t'_i.y$, there is always a tuple at $t'_i.y$ in any slab-files merged from S_i.

Let t_i^x denote the tuple, promoted from t_i, in *the x-th following merged slab-file*, that is, the resulting slab-file after x merging steps from S_i. By Lemma 9, t_i must be immediately below or in the same line as its next promoted tuple. Then we have:

$$t_i.y = t_i^0.y \leq t_i^1.y \leq t_i^2.y \leq ... \leq t_i^{\alpha-1} \leq t_i^{\alpha} = t_{max}.y. \quad (2)$$

Also, there is no tuple between t_i^x and t_i^{x+1} in the x-th following merged slab-file for any $x \in [0, \alpha - 1]$ by Lemma 9. This contradicts that, at $t'_i.y$, there exists a tuple in any following merged slab-file such that $t_i.y < t'_i.y \leq t_{max}.y$ by the initial assumption. □

The bottom-up process.

Now we are ready to actually construct max-regions. The basic processing scheme of constructing max-regions is similar to that of constructing slab-files in the first phase. Thus, in a bottom-up manner, we continuously merge region-files, and thereby the final region-file of the entire space will be created, which contains all the max-regions w.r.t. R. More specifically, each merging step is the process of finding intersections of the max-ranges in each region-file and the marked h-line in the upper level slab-file.

The details of each merging step, called *IntersectSweep*, are presented in Algorithm 6. IntersectSweep also uses a bottom-up sweeping scheme, and m region-files are given for the input, together with one merged slab-file, m slab-files, and the file of spanning rectangles. We use $upSum[i]$ as before, and specially maintain a pair of lists L_i and $actL_i$ for each region file X_i to keep track of active max-ranges in X_i, both of which are initially empty (Lines 1 - 2). When the sweep line encounters spanning rectangles, we update $upSum[i]$ values accordingly as before (Lines 4 - 7). Especially, when the sweep line encounters the bottom of any max-range, we insert the max-range to the appropriate list L_i (Lines 14 - 15). When the sweep line encounters a marked h-line in the merged slab-file, we find the intersections of this h-line and all the max-ranges in L_i's, and insert those max-ranges into the corresponding active list $actL_i$'s (Lines 21 - 24).

More precisely, we scan all the max-ranges currently residing in L_i's, and check if they intersect the sweeping line ℓ. If a max-range u is intersecting ℓ, we also check whether the weight of the tuple $t_i \in S_i$ that contributes to u's bottom edge plus the total weight of the current spanning rectangles equals the weight of the marked h-line t under consideration (i.e., $t.sum$) (Line 22). Then we update $u.y_1$ to be the y-coordinate of t, as pictured in Figure 18(a), assuming that u' is the next max-range to be returned in the next region-file (i.e., X) (Line 23). To activate u, we move u from L_i to $actL_i$, which indicates that u has ever met a marked h-line (Line 24). It is not difficult to notice that the purpose of this process is to determine the bottom boundary of the next max-range to be returned.

Now we should update the top boundaries of active max-ranges, which are residing in $actL_i$'s. We observe that the top of a max-range is changed only when encountering spanning rectangles. As shown in Figure 18(a), the original top of u should be changed to the top of u', which is the top of the lowest spanning rectangles involved in u. Thus,

whenever we encounter the top of a rectangle spanning slab γ_j, we examine each active max-range in the corresponding $actL_j$ and update its y_2 accordingly by comparing their original y_2 (Lines 8 - 10). Then each updated max-range is inserted into the next region-file to be returned (Line 11). Furthermore, the area cropped from original u may encounter a marked h-line later, and hence it is moved again to L_i from $actL_i$ (Lines 12 - 13).

However, we still have an issue to construct all the max-regions w.r.t. R, which is the case of *spanning max-ranges* as shown in Figure 18(b). Our strategy for this case is to use $upSum[i]$ values, which represent the total weight of all the rectangles that span the i-th slab at the current sweeping line, as before. If the current $upSum[i]$ value for a slab γ_i is equal to $t.sum$, then we construct the spanning max-range u_{span} (Lines 18 - 20). A more challenging problem in constructing u_{span} is to know the y-coordinate of the top of u_{span}. Fortunately, we can always know this y-coordinate by merging with the max-range created in L_{i-1}, which is supported by the following lemma:

LEMMA 12. *Let L_i be the set of active max-ranges in a region-file X_i, and u_i be a spanning max-range created in L_i. Then there should exist a merge-able max-range u_{i-1} created in L_{i-1}.*

PROOF. Let R_{span} be the set of spanning rectangles covering u_i, and r_o be the right-most one in R_{span}. Then there must be a slab γ_j that contains the left side edge of r_o, which means r_o does not span γ_j. Since r_o is the right-most rectangle in R_{span}, all other rectangles in R_{span} should either span or intersect γ_j. Therefore, there exists an intersecting area covered by all the rectangles in R_{span} in γ_j, and hence the area is also a max-range, denoted by u_j. Then u_j cannot span γ_j since r_o does not span γ_j. If $j = i - 1$, this lemma is proved, since it is obvious that u_j can be merge-able with u_i. If $j < i - 1$, we can also claim that there should be another spanning max-range u_{i-1}, since all the rectangles in R_{span} must span γ_{i-1}. It is not difficult to know that this u_{i-1} is also merge-able with u_i, since they consist of the same set of rectangles R_{span}. □

Note that this is feasible because we scan L_i's from left to right (i.e., $i = 1$ to m).

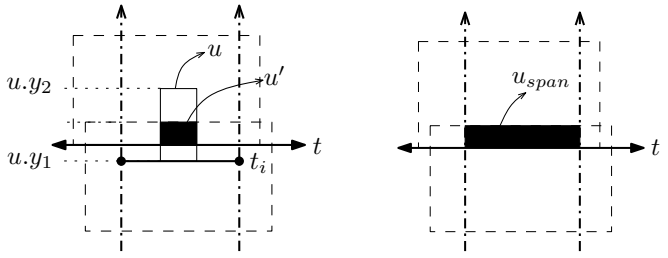


Fig. 18. The illustration of the IntersectSweep algorithm

Example 7. Figure 19 shows the final region-file containing all the final max-regions that are shown in Figure 14(a), which is merged from four region-files shown in Figure 17 considering the final slab-file shown in Figure 16(a). Each final max-region is represented as a black-filled rectangle, and max-ranges in four region-files are also represented here as empty rectangles.

When the sweeping line is at y_2, we find all the max-ranges intersecting the line since the h-line at y_2 in the final slab-file is marked. For each max-range intersecting the sweeping line y_2, we check if the weight of the max-range is equal to the weight of the marked h-line at y_2 in the final slab-file (i.e., 3). The weight of each max-range at

ALGORITHM 6: IntersectSweep

Input: a slab-file S of γ, m slab-files $S_1, ..., S_m$ and m region-files $X_1, ..., X_m$ for m sub-slabs of γ, a set of spanning rectangles R'
Output: a region-file X for γ. Initially $X \leftarrow \phi$

```
1  for i = 1 to m do
2      upSum[i] ← 0, L_i ← φ, actL_i ← φ
3  while sweeping the horizontal line ℓ from bottom to top do
4      if ℓ meets the bottom of r_o ∈ R' then
5          upSum[j] ← upSum[j] + w(o), ∀j s.t. r_o spans γ_j
6      if ℓ meets the top of r_o ∈ R' then
7          upSum[j] ← upSum[j] − w(o), ∀j s.t. r_o spans γ_j
8          foreach u ∈ actL_j, , ∀j s.t. r_o spans γ_j do
9              copy u into u'
10             u'.y_2 ← min{u'.y_2, ℓ.y}
11             output u' to X with merge
12             u.y_1 ← u'.y_2
13             insert u into L_i and delete u from actL_i
14     if ℓ meets the bottom of u_i ∈ X_i then
15         insert u_i into L_i
16     if ℓ meets t ∈ S, s.t. t.marked = true then
17         for i = 1 to m do
18             if upSum[i] = t.sum then
19                 generate a spanning max-range u_span on t.y
20                 insert u_span into actL_{i−1} with merge
21             foreach u ∈ L_i do
22                 if (u.y_1 ≤ t.y) ∧ (u.y_2 > t.y) ∧ (t_i.sum + upSum[i] = t.sum, s.t. t_i ∈ S_i is
                   promoted to t) then
23                     u.y_1 ← t.y
24                     insert u into actL_i and delete u from L_i
25 for i = 1 to m do
26     foreach u ∈ L_i ∪ actL_i do
27         output u to X with merge
28 return X
```

y_2 equals the weight of the h-line at the bottom of the max-range plus the total weight of the spanning rectangles at y_2. For the max-range at y_2 in the region-file of γ_3 (i.e., X_3), the weight of the underlying h-line (which is at y_0) is 1 as shown in Figure 17 and the total weight of spanning rectangles at y_2 in γ_3 is 2 as shown in Figure 16(b). Thus, the weight of the max-range at y_2 in X_3 is 3, and hence we update this max-range to generate a new max-range for the merged region-file. Note that the updated max-range in X_3 is a *piece* of the final max-region, because the merged region-file is indeed the region-file with respect to the entire space. Another max-range intersecting the sweeping line at y_2, that is, the max-range at y_2 in the region-file of γ_4 (i.e., X_4), is similarly processed and merged with the updated max-range in X_3, and thereby the final max-region is created as represented as a black-filled rectangle at y_2 in Figure 19.

Other final max-regions lying on the h-line at y_4 are created in a similar manner. Note that the top edge of the max-region at y_4 in γ_3 can be determined by the y-

coordinate of the top of the closest spanning rectangle, which is y_5 as shown in Figure 16(b).

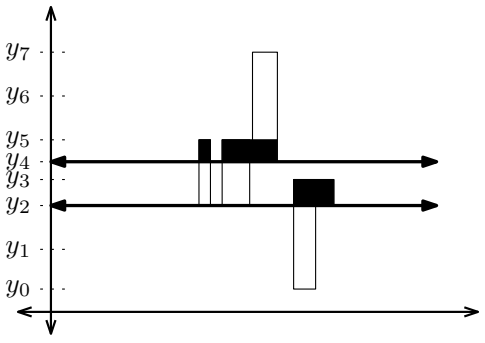


Fig. 19. The final region-file

LEMMA 13. *Let K be the number of rectangles in slab γ in a certain recursion, and T be the number of max-regions w.r.t. R residing in γ. Given a slab-file S of γ, m slab-files $S_1, ..., S_m$ and m region-files $X_1, ..., X_m$ of slabs $\gamma_1, ..., \gamma_m$, s.t., $\gamma = \cup_{i=1}^{m} \gamma_i$, and a set of spanning rectangles R', the IntersectSweep algorithm returns the region-file X of γ in $O((K+T)/B)$ I/O's.*

PROOF. Since IntersectSweep is also a sweeping algorithm, its I/O cost is proportional to the number of tuples in files. Therefore, by Lemma 11, we have $O(|S|/B + |S_1|/B + ... + |S_m|/B + |X_1|/B + ... + |X_m|/B + |R'|/B) = O((K+T)/B)$. □

8.2.4. Overall Algorithm. The overall algorithm of TwoPhaseMaxRS is shown in Algorithm 7. The inputs of TwoPhaseMaxRS are the set R of rectangles and the number of sub-slabs m, and the output is the region-file containing all the max-regions w.r.t. R. We use three global variables, $\mathcal{R}$, $\mathcal{S}$, and $\mathcal{R}'$, which are a set of sets of rectangles, a set of slab-files, and a set of sets of spanning rectangles, respectively. The algorithm consists of two subroutines, namely *DoFirstPhase* and *DoSecondPhase*. DoFirstPhase is almost the same as ExactMaxRS except that it uses PathSweep (Algorithm 4) instead of MergeSweep (Algorithm 1). After DoFirstPhase is finished, all the sets of files (i.e., $\mathcal{R}$, $\mathcal{S}$, and $\mathcal{R}'$) are built, and they are further used in DoSecondPhase.

ALGORITHM 7: TwoPhaseMaxRS

Input: a set of rectangles R, the number of sub-slabs m
Output: a region-file X with regard to the entire space
Global variables: a set of sets of rectangles $\mathcal{R}$, a set of slab-files $\mathcal{S}$, a set of sets of spanning rectangles $\mathcal{R}'$

```
1 γ ← a slab whose x-range is (−∞, ∞)
2 DoFirstPhase(R, γ, m)                          /* R, S, and R' are built. */
3 X ← DoSecondPhase(γ, m)
4 return X
```

The details of the recursive algorithm DoSecondPhase are shown in Algorithm 8. First, we obtain the slab-file S of the input slab γ from $\mathcal{S}$ (Line 1), and check if γ was not further split in the first phase. If γ has sub-slabs, say $\gamma_1, ..., \gamma_m$, we obtain all the slab-files of these sub-slabs and the file of spanning rectangles at the level of γ (Lines 2

- 4). With these slab-files and the file of spanning rectangles, we invoke the MarkSweep algorithm (Algorithm 5) (Line 5). Next, for each sub-slab γ_i, we recursively invoke DoSecondPhase, which returns the region-file of γ_i (Lines 6 - 7). Finally, we merge all these returned region-files by IntersectSweep (Algorithm 6), and thereby obtain the merged region-file X to be returned (Line 8).

In the case that γ is a slab at the bottom level of recursion, we initially create the region-file X of γ, with the file R of rectangles residing in γ and the corresponding slab-file S (Lines 9 - 11). Note that this process can be done in the main memory as γ is at the bottom level. The in-memory algorithm *CreateRegionFile* is straightforward, so the details are skipped here.

ALGORITHM 8: DoSecondPhase

Input: a slab γ, the number of sub-slabs m
Output: a region-file X for γ

```
1  S ← the slab-file of γ in 𝒮
2  if γ is not a slab at the bottom level of recursion then
3      S_1, ..., S_m ← the slab-files of sub-slabs of γ in 𝒮
4      R' ← the set of spanning rectangles of γ in 𝓡'
5      MarkSweep(S, S_1, ..., S_m, R')
6      for i = 1 to m do
7          X_i ← DoSecondPhase(γ_i, m)
8      X ← IntersectSweep(S, S_1, ..., S_m, X_1, ..., X_m, R')
9  else
10     R ← the set of rectangles of γ in 𝓡
11     X ← CreateRegionFile(S, R)                    /* in-memory algorithm */
12 return X
```

8.3. Theoretical Analysis

This section provides a theoretical analysis of TwoPhaseMaxRS in the perspective of its I/O efficiency and correctness.

8.3.1. I/O Efficiency. First, we prove the I/O cost of TwoPhaseMaxRS as the following theorem:

THEOREM 8. *The TwoPhaseMaxRS algorithm solves the AllMaxRS problem in* $O(((N+T)/B)\log_{M/B}(N/B))$ ***I/O's,*** *where* T *is the number of max-regions to be returned.*

PROOF. Let us first examine the first phase. It is easy to follow that the first phase requires exactly the same I/O's as ExactMaxRS, since PathSweep needs the same I/O cost as MergeSweep by Lemma 8.

In the second phase, IntersectSweep needs more I/O's than MergeSweep, which is proved in Lemma 13. The number of recursions, however, are still $O(\log_{M/B}(N/B))$. Thus, we have to find out the total I/O's required for each level. Since each max-region has two side edges, we have $\sum_{\forall i}|X_i| \leq 2T = O(T)$, which leads for the total I/O's for each level to be $O((N+T)/B)$ by Lemmas 10 and 13. Therefore, the total I/O cost of the second phase is $O(((N+T)/B)\log_{M/B}(N/B))$, which dominates the total I/O cost of TwoPhaseMaxRS. □

8.3.2. Correctness. Now we prove the correctness of TwoPhaseMaxRS by showing the soundness and completeness of the results returned from TwoPhaseMaxRS as the following theorem:

THEOREM 9. *The set of max-regions in the region-file returned from the TwoPhaseMaxRS algorithm is sound and complete with regard to a given dataset R.*

Completeness.
Let us establish the completeness first. The completeness of TwoPhaseMaxRS is identical to the argument that all the max-regions w.r.t. R must be contained in the final region-file returned from TwoPhaseMaxRS. This can be started with the following lemma:

LEMMA 14. *Let $t_i, t'_i \in S_i$ be two consecutive h-lines, where S_i is the slab-file of slab γ_i. If the bottom edge of a max-region ρ_i w.r.t. R residing in γ_i is on a horizontal line ℓ_i between t_i and t'_i (i.e., $t_i.y \leq \ell_i.y < t'_i.y$), there should be a max-range with the x-range the same as that of ρ_i whose bottom edge is on t_i.*

PROOF. Since t_i and t'_i are consecutive, we can guarantee that the max-intervals are not changed from t_i to t'_i, and the maximum location-weight in the rectangular area, denoted by r_{γ_i}, formed by t_i, t'_i, and the vertical boundaries of γ_i is $t_i.sum$ by the definition of a tuple in slab-files.

Let $t_{max} \in S$ be the marked h-line that contains the bottom edge of ρ_i (i.e., $\ell_i.y = t_{max}.y$), where S is the final slab-file of the entire space. Then we will prove that t_i is eventually marked by t_{max} after α marking steps, which implies that t_i is eventually promoted to t_{max} after α merging steps.

First of all, we can claim that the maximum location-weight in r_{γ_i} becomes $t_{max}.sum$ after all the merging steps. Let W^x_{span} be the total weight of spanning rectangles covering $t_{max}.y$ in the x-th merging step from S_i. Then we have:

$$t_i.sum + \sum_{x=1}^{\alpha} W^x_{span} = t_{max}.sum \tag{3}$$

We first examine the cases of $\alpha = 1$ and $\alpha = 2$, and then generalize to the case of $\alpha = k$.

Case of $\alpha = 1$. In this case, after one merging step, we obtain the final slab-file. By (3), $t_i.sum + W^1_{span} = t_{max}.sum$. Also, since t_i and t'_i are consecutive, t_i is immediate below or in the same line as t_{max}. Therefore, by Lemma 9, t_i is marked by t_{max}.

Case of $\alpha = 2$. In this case, two merging steps are performed to obtain the final slab-file. Let us first consider the merged slab-file, denoted by S^{next}, after one merging step. Then there must be a h-line in S^{next} whose location-weight is $t_i.sum + W^1_{span}$. Let y_1 be the y-coordinate of the h-line. Then we have $t_i.y \leq y_1 \leq t_{max}.y$.

Now we will prove that $t_i.sum + W^1_{span}$ is the maximum location-weight from y_1 to $t_{max}.y$ in S_{next}. Suppose, by contradiction, that there exists another tuple $t_j \in S_j$ at y_1 such that $t_j.sum > t_i.sum + W^1_{span}$. Then, after second merging step, the maximum location-weight at $t_{max}.y$ becomes $t_j.sum + W^2_{span}$, since spanning rectangles that cover the slab of S_i also cover the slab of S_j in the second merging step. By (3) and the initial assumption, we have $t_j.sum + W^2_{span} > t_i.sum + W^1_{span} + W^2_{span} = t_{max}.sum$, which is a contradiction that $t_{max}.sum$ is the maximum.

Since $t_i.sum + W^1_{span}$ is the maximum from y_1 in S^{next}, there must be a tuple $t^1_i \in S^{next}$ s.t. $t^1_i.sum = t_i.sum + W^1_{span}$ and $t^1_i.y = y_1$. Furthermore, if any other tuples between y_1 and $t_{max}.y$ exist, they should have $t^1_i.sum$ as their location-weight. Let t^1_{max} be the highest one among those tuples. Then we can derive that $t^1_{max}.sum = t^1_i.sum =$

$t_{max}.sum - W^2_{span}$ by (3). Therefore, t^1_{max} is marked by t_{max} by Lemma 9. Moreover, since $t_i.y \leq y_1 \leq t^1_{max}.y$ and $t_i.sum = t^1_{max}.sum - W^1_{span}$, t_i is also marked by t^1_{max}.

Case of $\alpha = k$. In this case, we generalize the case of $\alpha = 2$ by defining the following notations:

— S^x: the slab-file merged from S_i after x merging steps, e.g., $S^0 = S_i$ and $S^k = S$.
— y_x: the lowest y-coordinate of the h-line whose location-weight becomes $t_i.sum + \sum_{j=1}^{x} W^j_{span}$ after x merging steps s.t. $y_{x-1} \leq y_x \leq t_{max}.y$, e.g., $y^0 = t_i.y$ and $y^k = t_{max}.y$.
— t^x_{max}: the highest one among the set of tuples between y_x and $t_{max}.y$ whose location-weight are equally set to $t_i.sum + \sum_{j=1}^{x} W^j_{span}$, e.g., $t^0_{max} = t_i$ and $t^k_{max} = t_{max}$.

Then we prove this case by induction. The base cases of $k = 1$ and $k = 2$ follow the proofs in the above two cases of $\alpha = 1$ and $\alpha = 2$, respectively. Now we check whether t_i can be eventually marked by t_{max}, assuming that t_i can be eventually marked by t^{k-2}_{max}.

By the definition of t^{k-1}_{max} and (3), we have $t^{k-1}_{max}.sum = t_{max}.sum - W^k_{span}$, and there is no tuple between $t^{k-1}_{max}.y$ and $t_{max}.y$ in S^{k-1}. Therefore, t^{k-1}_{max} is marked by t_{max}.

Now we check whether t^{k-2}_{max} can be marked by t^{k-1}_{max}. Similar to t^{k-1}_{max}, we have $t^{k-2}_{max}.sum = t^{k-1}_{max}.sum - W^{k-1}_{span}$. Also, $t^{k-2}_{max}.y \leq t^{k-1}_{max}.y$ is guaranteed by the following two cases:

1) If $y_{k-1} \geq t^{k-2}_{max}.y$, then $t^{k-2}_{max}.y \leq y_{k-1} \leq t^{k-1}_{max}.y$.

2) If $y_{k-1} < t^{k-2}_{max}.y$, then there should be a tuple at $t^{k-2}_{max}.y$ in S^{k-1}, since its location-weight is also the maximum and it will cover a different set of max-intervals from the set of max-intervals covered by the tuple at y_{k-1} in S^{k-1} (which is the same set of max-intervals covered by the tuple at y_{k-2} in S^{k-2}). This is why $t^{k-2}_{max}.y$ exists in S^{k-2}. Since t^{k-1}_{max} is defined as the highest one among all the tuples between y_{k-1} and $t_{max}.y$, we have $t^{k-2}_{max}.y \leq t^{k-1}_{max}.y$. Therefore, t^{k-2}_{max} is also marked by t^{k-1}_{max}, because $t^{k-2}_{max}.y \leq t^{k-1}_{max}.y$ also implies that there is no tuple in S^{k-2} between $t^{k-2}_{max}.y$ and $t^{k-1}_{max}.y$ by the definition of t^x_{max}.

Now we can claim that t_i is eventually marked by t_{max} after α marking steps, and there should be a max-range whose bottom edge is on t_i in the region-file of γ_i. □

Indeed, Lemma 14 can be generally applicable to any slab-file in any level of recursion. Thus, we can guarantee that there is no missing max-region w.r.t. R, since any max-region can be constructed by its corresponding max-range containing its bottom edge. This concludes the proof of the completeness of TwoPhaseMaxRS.

Soundness.

It is not difficult to find out that TwoPhaseMaxRS is also sound. By Lemma 11, it is proved that every max-range in any slab-file should contain at least one max-region w.r.t. R. Furthermore, by IntersectSweep, we construct the max-ranges for the upper level of recursion, which are laid on only marked h-lines in the upper level slab-file, by checking their location-weights. Since it is guaranteed that each marked h-line in any slab-file is eventually promoted to the marked h-line in the final slab-file, we can finally construct max-ranges whose bottom edges are on the marked h-line in the final slab-file and their location-weight is obviously the same as the maximum location-weight of the h-line. This implies that each max-range in the final region-file is indeed a max-region w.r.t. R, since every point in the max-range has the same location-weight, by Definition 9, which is the global maximum. The proof of the soundness of TwoPhaseMaxRS is done.

9. EMPIRICAL STUDY

In this section, our objective is to evaluate the practical performance of the proposed external-memory algorithms, particularly focusing on the I/O cost.

9.1. Environment Setting

9.1.1. Datasets. We use both real and synthetic datasets in the experiments. We first generate synthetic datasets under uniform distribution and Gaussian distribution. We set the cardinalities of dataset (i.e., $|O|$) to be from 100,000 to 500,000 (default 250,000). The range of each coordinate is set to be $[0, 4|O|]$ (default $[0, 10^6]$).

We also use two real datasets, namely *North East (NE) dataset* and *GeoLife (GEO) dataset*. The NE dataset contains 123,593 postal addresses in New York, Philadelphia, and Boston[3], which can be regarded as an example of our intuitive application of MaxRS (recall the example of finding the best location of a new pizza store in Section 1). The GEO dataset is a set of 24,858,308 GPS locations collected from about 200 mobile subscribers for several years in the *GeoLife* project of Microsoft Research Asia[4]. Based on the average location, we discard the points that are extremely far away from the average, and thereby the resulting cardinality is 11,862,976. This dataset is a representative example of a massive historical dataset, and useful to evaluate the performance of our algorithms in a more scalable scenario. The range of coordinates of NE and GEO are normalized to $[0, 10^6]$ and $[0, 10^9]$, respectively. The specification of two real datasets are listed in Table IV.

9.1.2. Competitors. Since no method is directly applicable to the MaxRS problem in spatial databases, we should externalize the in-memory algorithm [Imai and Asano 1983; Nandy and Bhattacharya 1995] for max-rectangle enclosing problem, described in Section 4, to be compared with our ExactMaxRS algorithm. In fact, the externalization of this in-memory algorithm is already proposed by Du et al. [Du et al. 2005], which is originally invented for processing their optimal-location queries. They present two algorithms based on plane-sweep, called *Naive Plane Sweep* and *aSB-Tree*, which are also applicable to the MaxRS problem, even though their main algorithm based on a preprocessed structure, called the *Vol-Tree*, cannot be used in the MaxRS problem. Both Naive Plane Sweep and aSB-Tree are basically the same as the in-memory algorithm in Section 4 except that they use the external-memory structure instead of the binary tree. Specifically, Naive Plane Sweep uses a simple sequential file, and aSB-Tree exploits a B-tree-like structure.

For the MaxCRS problem, we compare one state-of-the-art algorithm, called *MaxOverlap* [Wong et al. 2009], with ApproxMaxCRS. MaxOverlap is originally intended to solve the *MaxBRNN* problem by Wong et al. However, its basic processing scheme is to find the most overlapping region in the arrangement of circles, and hence it can be applied to the MaxCRS problem as well.

9.1.3. Performance Metrics. As performance metrics, we first use the number of I/O's, precisely the number of transferred blocks during the process, which is the main issue dealt with in this article. We also report all the corresponding execution times, aiming to reveal whether our algorithms are also efficient in terms of the CPU performance.

9.1.4. Parameters. We fix the block size to 4KB, and set the buffer size to 1024KB by default. The GEO dataset deserves a larger buffer size, so we set the default size to 4096KB. (recall that we consider a massive dataset which cannot be fully loaded into the main memory). Also, for MaxRS and AllMaxRS, we set the rectangle size to

[3]http://www.rtreeportal.org
[4]http://research.microsoft.com/en-us/projects/geolife

Table IV. The specifications of real datasets

Dataset	Cardinality	Space size
NE	123,593	$1M \times 1M$
GEO	11,862,976	$1G \times 1G$

Table V. The default values of parameters

Parameter	Default value
Cardinality ($\|O\|$)	250,000
Block size	4KB
Buffer size	1024KB, 4096KB (GEO)
Space size	$1M \times 1M$
Rectangle size ($d_1 \times d_2$)	$1K \times 1K$
Circle diameter (d)	$1K$

1000×1000 by default. Similarly, for the MaxCRS problem, we set the circle diameter to 1000 by default. All the default values of parameters are presented in Table V.

The way of determining the number of slabs (i.e., m) deserves a bit more explanation. As mentioned earlier, m is basically set to $\Theta(M/B)$, that is, the number of blocks in the main memory. However, a larger m does not always imply a better performance. The I/O complexity of our algorithms commonly have two factors. The first factor is either $O(N/B)$ or $O((N+T)/B)$, and the second factor is $O(\log_m (N/B))$. Intuitively, the first factor explains the I/O cost of one recursion, and the second factor indicates the total number of recursions. Once the number of recursions gets less than a certain integer, say i, m does not have to be large until the number of recursions becomes $i-1$. Rather, a lager m can cause additional overhead due to the occurrence of many spanning rectangles unless it does not reduce the number of recursions. Based on this heuristic, we set m to be the minimum value such that $\lfloor \log_m (N/B) \rfloor$ is maximized.

9.1.5. Environment. We implement all the algorithms in Java, and conduct all the experiments on a PC running Linux (Ubuntu 13.10) equipped with Intel Core i7 CPU 3.4GHz and 16GB memory.

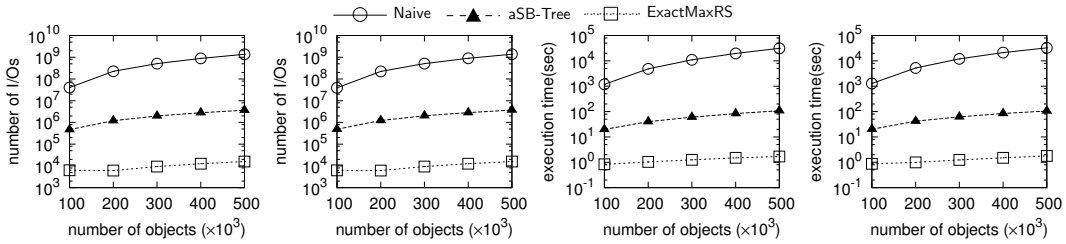
9.2. Experimental Results on MaxRS

In this section, we present our experimental results on the MaxRS problem. By varying the parameters, we examine the performance of alternative algorithms in terms of the I/O cost and execution time. Note that both the I/O cost and execution time are in log scale in all the relevant graphs in this section.

9.2.1. Effect of the Dataset Cardinalities. Figure 20 shows the experimental results for varying the total number of objects in the dataset. Both of the results of Gaussian distribution and uniform distribution show that our ExactMaxRS is much more efficient than the algorithms based on plane-sweep. Especially, even if the dataset gets larger, the ExactMaxRS algorithm achieves performance similar to that on the smaller dataset, which effectively shows that our algorithm is scalable to datasets of a massive size.

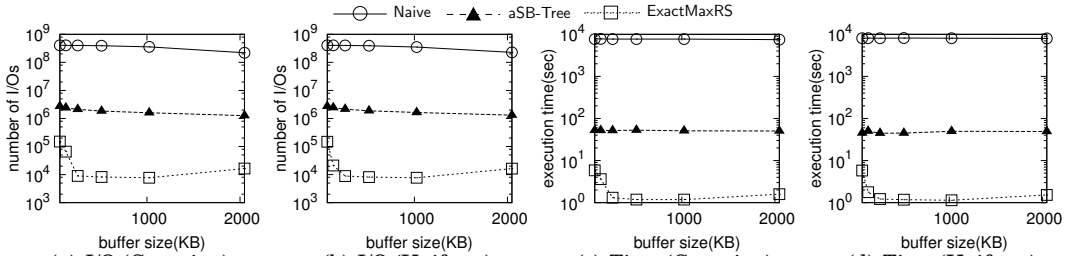
Considering the overall performance (together with the CPU time), ExactMaxRS is extremely faster than Naive Plane Sweep and aSB-Tree, and its execution time is always about just one second.

9.2.2. Effect of the Buffer Size. Figure 21 shows the experimental results for varying the buffer size. Even though all the algorithms exhibit better performance as the buffer size increases, the ExactMaxRS algorithm is more sensitive to the size of buffer than the others. This is because our algorithm uses the buffer more effectively. As proved in Theorem 2, the I/O complexity of ExactMaxRS is $O((N/B)\log_{M/B}(N/B))$, which means the larger M, the smaller the factor $\log_{M/B}(N/B)$. Nevertheless, once the buffer size is larger than a certain size, the ExactMaxRS algorithm also shows behavior similar to the others, since the entire I/O cost will be dominated by $O(N/B)$, i.e., linear cost.



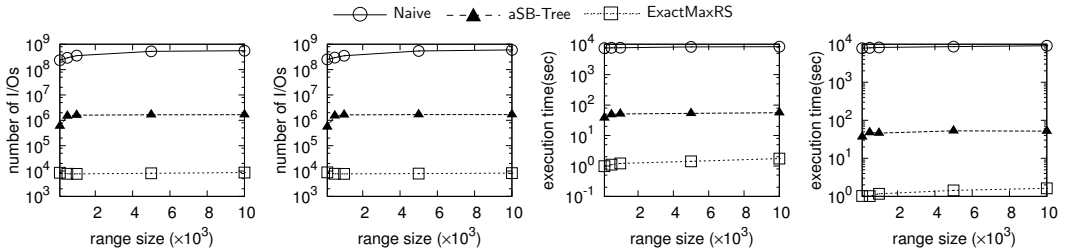
(a) I/O (Gaussian) (b) I/O (Uniform) (c) Time (Gaussian) (d) Time (Uniform)

Fig. 20. Effect of the dataset cardinalities using synthetic datasets on MaxRS



(a) I/O (Gaussian) (b) I/O (Uniform) (c) Time (Gaussian) (d) Time (Uniform)

Fig. 21. Effect of the buffer size using synthetic datasets on MaxRS



(a) I/O (Gaussian) (b) I/O (Uniform) (c) Time (Gaussian) (d) Time (Uniform)

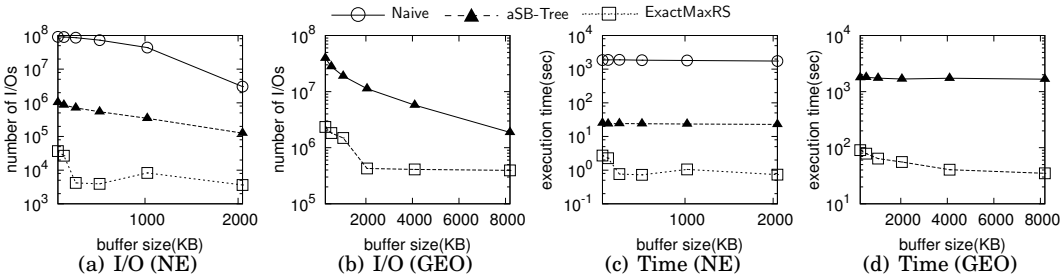
Fig. 22. Effect of the range size using synthetic datasets on MaxRS

The graphs representing execution times are similar to those of I/O costs, which means that the overall performance of these external-memory algorithms are well explained by the I/O cost.

9.2.3. Effect of the Range Size. Figure 22 shows the experimental results for varying the range parameters. Without loss of generality, we use the same value for each dimension, i.e., each rectangle is a square. It is observed that the ExactMaxRS algorithm is less influenced by the size of range than the other algorithms. This is because as the size of range increases, the probability that rectangles overlap also increases in the algorithms based on plane-sweep, which means that more intervals should be inserted into the maintaining data structure such as the B-tree. Meanwhile, the ExactMaxRS algorithm is not much affected by the overlapping probability.

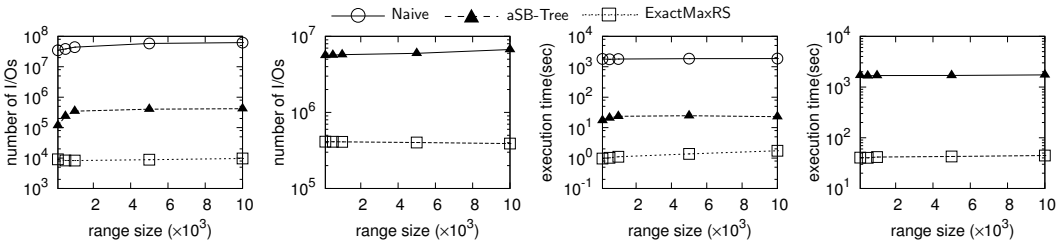
9.2.4. Results of Real Datasets. We conduct the same kind of experiments on real datasets except varying cardinalities. As shown in Table IV, since GEO is much larger than NE, we use larger buffer sizes for GEO.

Overall trends of the graphs are similar to the results in synthetic datasets, as shown in Figures 23 and 24. Note that in Figures 23(b), 23(d), 24(b), and 24(d), Naive Plane



(a) I/O (NE) (b) I/O (GEO) (c) Time (NE) (d) Time (GEO)

Fig. 23. Effect of the buffer size using real datasets on MaxRS



(a) I/O (NE) (b) I/O (GEO) (c) Time (NE) (d) Time (GEO)

Fig. 24. Effect of the range size using real datasets on MaxRS

Sweep is not presented. This is because the execution time of the Naive Plane Sweep algorithm when using GEO is extremely long[5], and therefore we cannot conduct such experiments for Naive Plane Sweep using GEO.

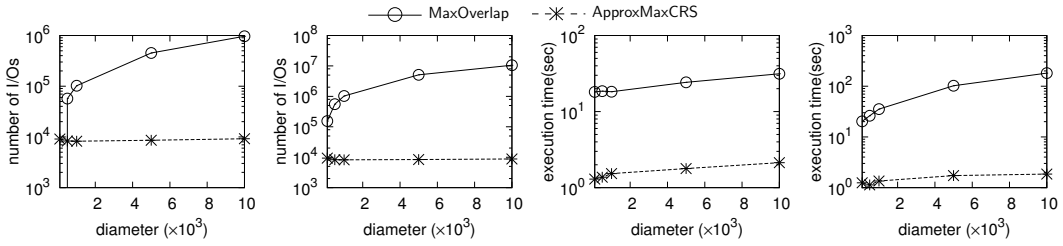
9.3. Experimental Results on MaxCRS

In this section, we present the experimental results with regard to the MaxCRS problem. We first show the performance comparison result in Section 9.3.1, and examine the quality of approximation of ApproxMaxCRS in Section 9.3.2. The GEO dataset is not used in this section, since it is too large to find the exact solution of the MaxCRS problem.

9.3.1. Performance Comparison. As mentioned earlier, we employ the MaxOverlap algorithm [Wong et al. 2009], which can exactly solve the MaxCRS problem, as a competitor of the ApproxMaxCRS algorithm. Let us illustrate the overall process of MaxOverlap briefly. The MaxOverlap algorithm comprises of three steps. First, the R-tree on the set of objects (i.e., O) should be built. Second, by using the R-tree on O, the algorithm performs N range queries, one from the location of each object in O, to construct a table having the overlapping relationship among the circles centered at objects in O (i.e., C). Finally, while scanning the table, the final answer can be obtained.

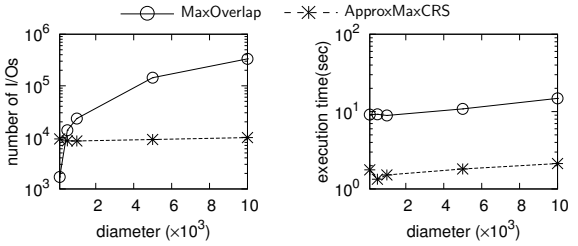
For the simplicity, among three steps of MaxOverlap, we only implement and measure the I/O cost and the execution time of the second step since the costs of other steps are similar to or less than that of the second step according to the analysis in [Wong et al. 2009]. Nevertheless, as shown in Figures 25 and 26, ApproxMaxCRS outperforms MaxOverlap in the experiments for varying the diameter. Especially when the diameter becomes large, the probability that circles overlap each other increases, which results in excessive I/O's of MaxOverlap.

[5]It took a few days for Naive Plane Sweep to find a single solution from the GEO dataset.



(a) I/O (Gaussian) (b) I/O (Uniform) (c) Time (Gaussian) (d) Time (Uniform)

Fig. 25. Performance comparison using synthetic datasets on MaxCRS



(a) I/O (NE) (b) Time (NE)

Fig. 26. Performance comparison using NE on MaxCRS

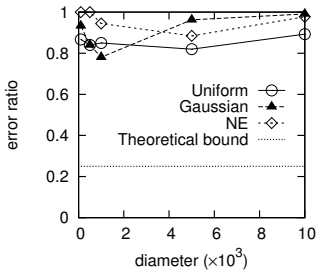


Fig. 27. Approximation quality

One interesting point lies in the case of a tiny diameter, in which the MaxOverlap algorithm causes only a few I/O's. This is because, in this case, MaxOverlap will visit only a small number of non-leaf nodes at high levels in the R-tree since circles with a tiny diameter rarely overlap each other. However, its execution time is longer than that of ApproxMaxCRS (even though it gets shorter for a smaller diameter). This can be interpreted that it is not very cheap to check the entries in such a few non-leaf nodes in the main memory to see whether each entry intersects the queried range (i.e., a circle centered at an object).

9.3.2. Quality of Approximation. Next, we evaluate the quality of approximation obtained from the ApproxMaxCRS algorithm in Figure 27. Since the quality can be different when the diameter changes, we examine the quality by varying d on both synthetic and real datasets. Optimal answers are obtained by implementing a theoretical algorithm [Drezner 1981] that has time complexity $O(n^2 \log n)$ (and therefore, is not practical). We observe that when the diameter gets larger, the quality of approximation becomes higher and more stable, since more objects are included in the given range. Even though theoretically our ApproxMaxCRS algorithm guarantees the $(1/4)$-approximation bound, the average approximation ratio is much larger than $1/4$ in practice, which is close to 0.9.

9.4. Experimental Results on AllMaxRS

In order to evaluate the proposed algorithms for the AllMaxRS problem, we conduct the same kind of experiments as done in Section 9.2.

Alternative algorithms that are compared with ExactMaxRS are not considered any more in this section, because their performance for finding only one max-region is even worse than that of our algorithms, SimpleAllMaxRS and TwoPhaseMaxRS, for returning all the max-regions. The I/O cost of ExactMaxRS is only provided as a bottom-line reference for assessing performance, even though it also returns only one max-region.

Although it is theoretically proved that TwoPhaseMaxRS is more efficient than SimpleAllMaxRS, the I/O complexity of TwoPhaseMaxRS includes some hidden constants. Furthermore, if the number of tied max-intervals is small enough (almost one as in ExactMaxRS), the cost of SimpleAllMaxRS will be close to that of ExactMaxRS. Thus, there could be cases where SimpleAllMaxRS's I/O cost is smaller than that of TwoPhaseMaxRS. From this observation, we can expect that SimpleAllMaxRS could be also useful in practice.

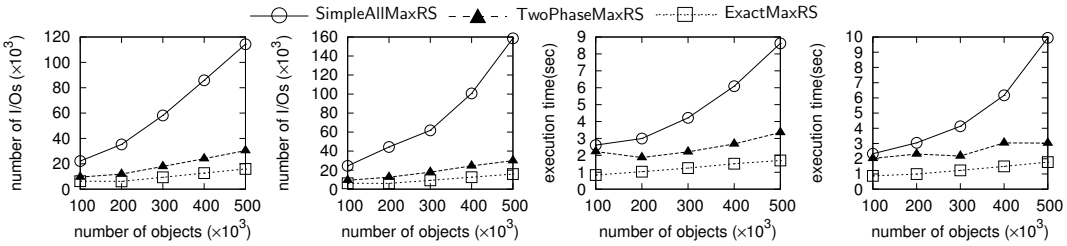
9.4.1. Effect of the Dataset Cardinalities. Figure 28 shows the results for varying the number of objects in the dataset. Unlike our expectation that SimpleAllMaxRS can be comparable to TwoPhaseMaxRS, in both of the distributions, TwoPhaseMaxRS is much more scalable than SimpleAllMaxRS. The main reason behind this result is that the second phase of TwoPhaseMaxRS needs only a small number of I/O's compared with the first phase, which will be shown in Section 9.4.5. Moreover, the total number of I/O's in the first phase is much less than the number of I/O's performed in SimpleAllMaxRS. This is obviously because TwoPhaseMaxRS maintains only one tuple per h-line, while all tied max-intervals at each h-line are maintained in SimpleAllMaxRS.

The graphs on the execution time show similar patterns to those on the I/O cost. A minor drawback of TwoPhaseMaxRS is that its CPU overhead seems relatively higher than SimpleAllMaxRS, as shown that the performance gap in Figures 28(c) and 28(d) is smaller than that in Figures 28(a) and 28(b).

9.4.2. Effect of the Buffer Size. Figure 29 shows the results for varying the buffer size. All the algorithms show very sensitive results to the buffer size, since they are basically rooted on ExactMaxRS. In some cases, the I/O cost even increases a little when the buffer size increases. This is due to the inaccuracy of our simple heuristic to determine the optimal number of slabs m. In practical environment, m can be appropriately tuned for further improving the performance. In a macro view, however, we can claim that the larger buffer size, the better performance, especially assuming that the cardinality of the dataset is massive (i.e., $N \gg M$).

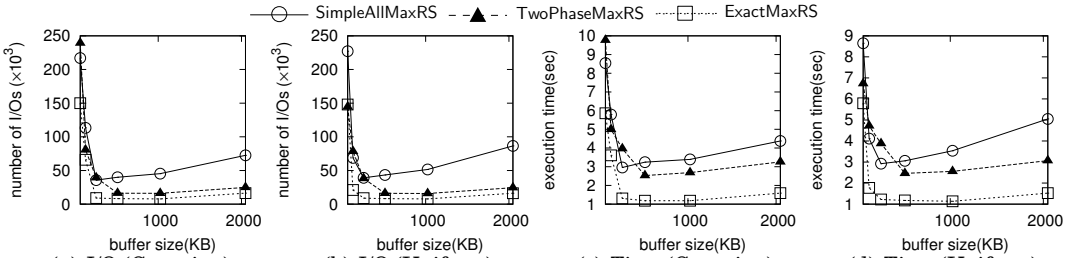
From the observation that the difference between total I/O's is not very large when the size of buffer is tiny, we conclude that TwoPhaseMaxRS tends to require more buffers than SimpleAllMaxRS. This is because, in the second phase of TwoPhaseMaxRS, the IntersectSweep algorithm needs to scan many kinds of files simultaneously, and each file requires at least one block for its I/O buffer. Nevertheless, TwoPhaseMaxRS shows almost always better results than SimpleAllMaxRS, and its overall trends are very close to ExactMaxRS which can be regarded as optimal. As with the result in Figure 28, the performance gap gets a little smaller in the result on the execution time, even though the overall trends are not changed much.

9.4.3. Effect of the Range Size. Figure 30 shows the results for varying the size of rectangle. For all the experimental results, the number of I/O's of TwoPhaseMaxRS is always less than that of SimpleAllMaxRS. One somewhat surprising result is that SimpleAllMaxRS shows extremely unsatisfactory result when the range size is very small. In this case, the count of each max-region is relatively small (less than 4), and hence the probability of occurring multiple max-regions with the same maximum count is also relatively high. Thus, there are not only many max-regions to be returned but also even more tied max-intervals to be maintained such that their counts are a little less than that of the final max-regions (e.g., 2 or 3). This demonstrates why SimpleAllMaxRS can be much inefficient in the worst case. Meanwhile, TwoPhaseMaxRS shows very stable results. The I/O cost and the execution time of TwoPhaseMaxRS are always within a constant factor of those of ExactMaxRS.



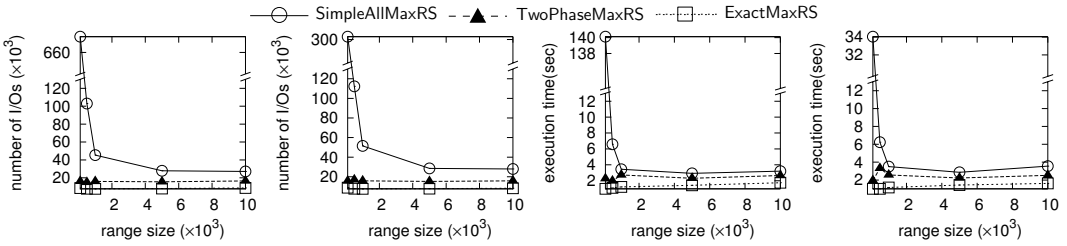
(a) I/O (Gaussian) (b) I/O (Uniform) (c) Time (Gaussian) (d) Time (Uniform)

Fig. 28. Effect of the dataset cardinalities using synthetic datasets on AllMaxRS



(a) I/O (Gaussian) (b) I/O (Uniform) (c) Time (Gaussian) (d) Time (Uniform)

Fig. 29. Effect of the buffer size using synthetic datasets on AllMaxRS

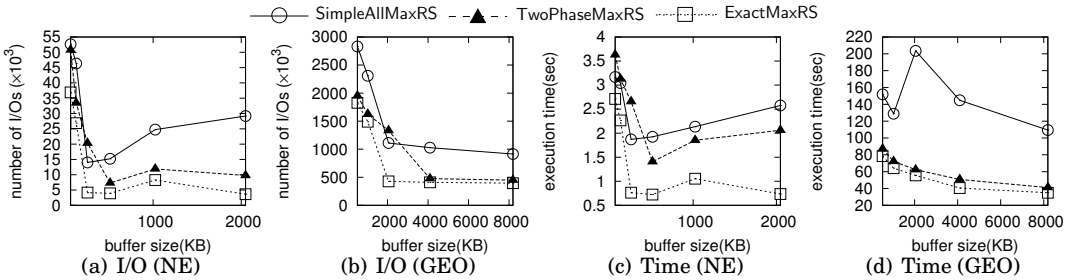


(a) I/O (Gaussian) (b) I/O (Uniform) (c) Time (Gaussian) (d) Time (Uniform)

Fig. 30. Effect of the range size using synthetic datasets on AllMaxRS

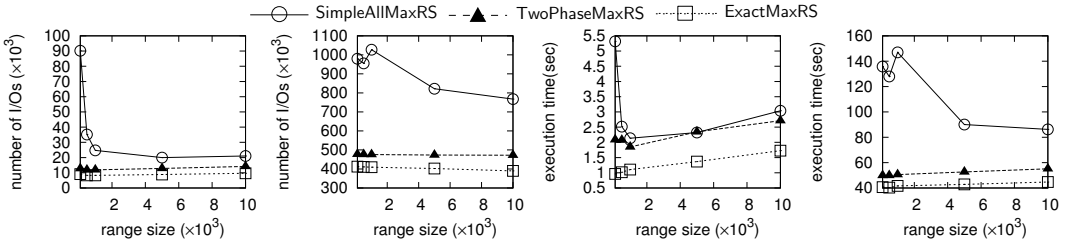
9.4.4. Results of Real Datasets. We also conduct the performance test using real datasets. Similarly, TwoPhaseMaxRS also shows more stable and better results than SimpleAllMaxRS. Furthermore, compared with the results using synthetic datasets, the fluctuation of the performance of SimpleAllMaxRS gets even higher. This shows that the cases unfavorable for SimpleAllMaxRS can arise even more in practice, which implies that TwoPhaseMaxRS is more efficient practically as well as theoretically than SimpleAllMaxRS.

9.4.5. Overall Performance Comparison. Overall performance on AllMaxRS is summarized in Figure 33. The graphs show the average ratio of the I/O cost and the execution time of each algorithm to those of ExactMaxRS. This result well explains why TwoPhaseMaxRS is much more efficient than SimpleAllMaxRS. Both in terms of the I/O cost and CPU time, the first phase of TwoPhaseMaxRS has almost the same performance as ExactMaxRS, and the second phase adds only a small amount of overhead, which is far less than the cost of ExactMaxRS as well as the first phase. Also, the cost of the second phase is related to the number of max-regions to be finally returned. For example, the GEO dataset almost always has only one max-region while the NE dataset has about 4 max-regions on average and 14 max-regions at most. Therefore,



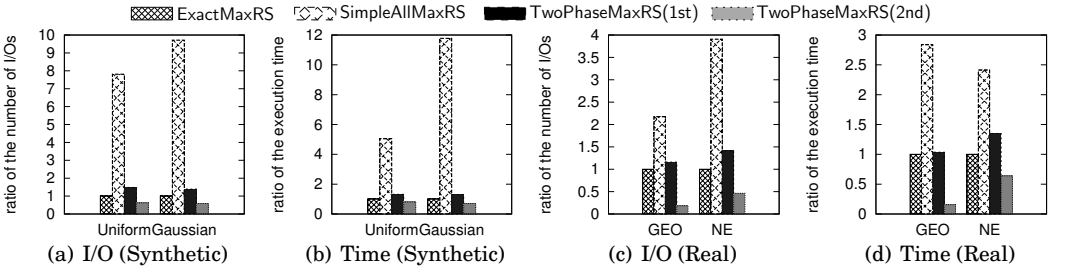
(a) I/O (NE) (b) I/O (GEO) (c) Time (NE) (d) Time (GEO)

Fig. 31. Effect of the buffer size using real datasets on AllMaxRS



(a) I/O (NE) (b) I/O (GEO) (c) Time (NE) (d) Time (GEO)

Fig. 32. Effect of the range size using real datasets on AllMaxRS



(a) I/O (Synthetic) (b) Time (Synthetic) (c) I/O (Real) (d) Time (Real)

Fig. 33. Overall performance comparison on AllMaxRS

the result shows that the overall cost of the second step in NE is relatively larger than that in GEO.

10. CONCLUSIONS

In this article, we solve the MaxRS problem in spatial databases. This problem is useful in many scenarios such as finding the most profitable service place and finding the most serviceable place, where a certain size of range should be associated with the place. For the MaxRS problem, we propose the first external-memory algorithm, ExactMaxRS, with a proof that the ExactMaxRS algorithm correctly solves the MaxRS problem in optimal I/O's. Furthermore, we propose an approximation algorithm, ApproxMaxCRS, for the MaxCRS problem that is a circle version of the MaxRS problem. We also prove that the ApproxMaxCRS algorithm gives a $(1/4)$-approximate solution to the exact solution for the MaxCRS problem. Through extensive experiments on both synthetic and real datasets, we demonstrate that the proposed algorithms are also efficient in practice.

By further extending the MaxRS problem, we propose a more complete version of the MaxRS problem, the AllMaxRS problem, which has not been studied even in the theoretical communities. We devise two external-memory algorithms, SimpleAllMaxRS

and TwoPhaseMaxRS, which are a simple extension of ExactMaxRS and a two-phase output-sensitive algorithm, respectively. Based on the extensive theoretical and experimental analysis, we prove that TwoPhaseMaxRS is much more efficient than SimpleAllMaxRS in both theory and practice.

In closing, we discuss some future works on MaxRS and its variants. First, we can think of the MaxRS problem in a higher dimensional space, extending this article that has been focused on *spatial* databases (i.e., objects are points in the 2D space). MaxRS and its variants in a higher dimensionality will also be useful in many applications involving the density analysis in data mining. Another future work can be devising an online algorithm for MaxRS. In this case, the goal is to preprocess a given set of objects in such a way that individual MaxRS queries with different parameters can be answered more quickly.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

We would like to thank the editors and anonymous reviewers for their helpful comments. Dong-Wan Choi and Chin-Wan Chung were supported in part by the National Research Foundation of Korea(NRF) Grant funded by the Korean Government(MSIP)(No. NRF-2014R1A1A2002499) and in part by Defense Acquisition Program Administration and Agency for Defense Development under the contract UD110006MD, Korea. Yufei Tao was supported in part by projects GRF 4165/11, 4164/12, and 4168/13 from HKRGC.

REFERENCES

Lars Arge, Mikael Knudsen, and Kirsten Larsen. 1993. A general lower bound on the I/O-complexity of comparison-based algorithms. In *Proceedings of Algorithms and Data Structures (WADS)*. 83–94.

Boris Aronov and Sariel Har-Peled. 2005. On approximating the depth and related problems. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 886–894.

Gill Barequet, Matthew Dickerson, and Petru Pau. 1997. Translating a convex polygon to contain a maximum number of points. *Computational Geometry* 8, 4 (1997), 167–179.

Mark De Berg, Sergio Cabello, and Sariel Har-Peled. 2009. Covering many or few points with unit disks. *Theory Comput. Syst.* 45, 3 (2009), 446–469.

B. M. Chazelle and D. T. Lee. 1986. On a circle placement problem. *Computing* 36, 1 (1986), 1–16.

Hyung-Ju Cho and Chin-Wan Chung. 2007. Indexing range sum queries in spatio-temporal databases. *Information & Software Technology* 49, 4 (2007), 324–331.

Dong-Wan Choi, Chin-Wan Chung, and Yufei Tao. 2012. A scalable algorithm for maximizing range sum in spatial databases. *PVLDB* 5, 11 (2012), 1088–1099.

Rezaul Alam Chowdhury and Vijaya Ramachandran. 2006. Cache-oblivious dynamic programming. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 591–600.

Matthew Dickerson and Daniel Scharstein. 1998. Optimal placement of convex polygons to maximize point containment. *Computational Geometry* 11, 1 (1998), 1–16.

Zvi Drezner. 1981. Note—On a modified one-center model. *Management Science* 27, 7 (1981), 848–851.

Yang Du, Donghui Zhang, and Tian Xia. 2005. The optimal-location query. In *International Symposium of Advances in Spatial and Temporal Databases (SSTD)*. 163–180.

Michael T. Goodrich, Jyh-Jong Tsay, Darren Erik Vengroff, and Jeffrey Scott Vitter. 1993. External-memory computational geometry (preliminary version). In *Proceedings of Annual Symposium on Foundations of Computer Science (FOCS)*. 714–723.

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), 310–323.

Marcus Jürgens and Hans-Joachim Lenz. 1998. The RA*-tree: an improved R-tree with materialized data for supporting range queries on OLAP-data. In *DEXA Workshop*. 186–191.

Iosif Lazaridis and Sharad Mehrotra. 2001. Progressive approximate aggregate queries with a multi-resolution Tree Structure. In *SIGMOD Conference*. 401–412.

Subhas C. Nandy and Bhargab B. Bhattacharya. 1995. A unified algorithm for finding maximum and minimum object enclosing rectangles and cuboids. *Computers and Mathematics with Applications* 29, 8 (1995), 45–61.

Dimitris Papadias, Panos Kalnis, Jun Zhang, and Yufei Tao. 2001. Efficient OLAP operations in spatial data warehouses. In *International Symposium of Advances in Spatial and Temporal Databases (SSTD)*. 443–459.

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

Cheng Sheng and Yufei Tao. 2011. New results on two-dimensional orthogonal range aggregation in external memory. In *Proceedings of ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. 129–139.

Shreyash Srivastava, Shaurya Ahuja, and Ankush Mittal. 2011. Determining most visited locations based on temporal grouping of GPS data. In *Proceedings of the International Conference on Soft Computing for Problem Solving (SocPros)*. 63–72.

Shivendra Tiwari and Saroj Kaushik. 2012. Extracting region of interest (ROI) details using LBS infrastructure and web-databases. In *Proceedings of the 13th IEEE International Conference on Mobile Data Management (MDM)*. 376–379.

Raymond Chi-Wing Wong, M. Tamer Özsu, Philip S. Yu, Ada Wai-Chee Fu, and Lian Liu. 2009. Efficient method for maximizing bichromatic reverse nearest neighbor. *PVLDB* 2, 1 (2009), 1126–1137.

Tian Xia, Donghui Zhang, Evangelos Kanoulas, and Yang Du. 2005. On computing top-t most influential spatial sites. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 946–957.

Xiaokui Xiao, Bin Yao, and Feifei Li. 2011. Optimal location queries in road network databases. In *Proceedings of International Conference on Data Engineering (ICDE)*. 804–815.

Man Lung Yiu, Xiangyuan Dai, Nikos Mamoulis, and Michail Vaitis. 2007. Top-k spatial preference queries. In *Proceedings of International Conference on Data Engineering (ICDE)*. 1076–1085.

Donghui Zhang, Yang Du, Tian Xia, and Yufei Tao. 2006. Progressive computation of the min-dist optimal-location query. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 643–654.

Yu Zheng, Lizhu Zhang, Xing Xie, and Wei-Ying Ma. 2009. Mining interesting locations and travel sequences from GPS trajectories. In *Proceedings of the 18th International Conference on World Wide Web (WWW)*. 791–800.

Zenan Zhou, Wei Wu, Xiaohui Li, Mong-Li Lee, and Wynne Hsu. 2011. MaxFirst for MaxBRkNN. In *Proceedings of International Conference on Data Engineering (ICDE)*. 828–839.

Online Appendix to: Maximizing Range Sum in External Memory

DONG-WAN CHOI, KAIST
CHIN-WAN CHUNG, KAIST
YUFEI TAO, Chinese University of Hong Kong and KAIST

A. PROOF OF LEMMA 7

To prove Lemma 7, we will use a new notion, called the *stair*, which is an important concept to examine the properties of the max-region, and several supporting lemmas, namely Lemmas A1, A2, and A3.

Definition A1 (*Stair*). Given $r \in R$, a stair of r is a pair of perpendicular rays that start from the same position in r and extend to a boundary of r.

Intuitively, stairs of r are the intersecting parts of other rectangles that intersect r (see Figure A1), which have the following properties:

Property 1 There are four types of stairs based on their directions, which are called the *up-left-stair*, *up-right-stair*, *down-left-stair*, and *down-right-stair*. To refer to both of the up-left-stair and down-left-stair, we use the term *left-stair*, and also use the terms *right-stair*, *up-stair*, and *down-stair* similarly. For example, in Figure A1, s_1 is an up-right-stair, and s_2 is a down-left-stair. We also regard a pair of adjacent edges of r as a stair.

Property 2 Each stair of r covers the part of the entire area of r from its starting position to the boundaries of r towards its directions.

Property 3 Based on the starting position and directions of a stair, the count increases by 1 when crossing the stair in the same direction; on the contrary, the count decreases by 1 when crossing the stair in the opposite direction.

By using the properties of stairs, we show the following lemma about a max-region inside a rectangle:

LEMMA A1. *Let ρ_k denote a max-region inside a rectangle $r \in R$. Then the horizontal boundaries of ρ_k consist of a pair of a left-stair and a right-stair that are facing each other, and also the vertical boundaries of ρ_k are a pair of an up-stair and a down-stair that are facing each other.*

PROOF. By the definition of the max-region, every max-region has the maximum count. Thus, when we cross the boundaries of a max-region in a way of escaping from the max-region, the count should be decreased. This implies that there cannot exist a max-region having either two left-stairs or two right-stairs as its side edges by Property 1 and Property 3. A similar rule is applied to the top and bottom edges of the max-region. □

It should be noted that Lemma A1 does not mean that each max-region should be bounded by four different stairs, e.g., a down-left stair, a down-right stair, and an up-right stair can construct a max-region.

LEMMA A2. *Let ℓ_h denote a horizontal line segment in r which has the same length as the width of r. Then the following statements are valid:*

DOI: http://dx.doi.org/10.1145/0000000.0000000

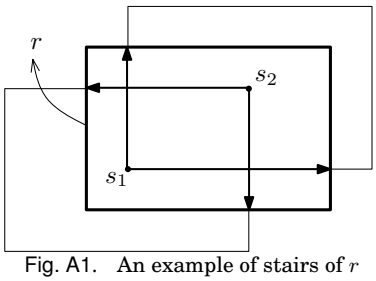


Fig. A1. An example of stairs of r

— *there exist at most k max-regions that intersect ℓ_h, and*
— *there exist at most k left-stairs and k right-stairs that intersect ℓ_h.*

PROOF. To prove the first statement by contradiction, suppose that there exist k' max-regions that intersect ℓ_h for some $k' > k$. By Lemma A1, there should be at least k' right-stairs and k' left-stairs, which are intersected with ℓ_h, for k' max-regions on ℓ_h. By Property 2, there should exist a position on ℓ_h covered by either all the right-stairs or all the left-stairs. This means that there is a position whose count is $k' > k$, which contradicts that the count of each max-region is k. Similarly, to prove the second statement, suppose that either k' left-stairs or k' right-stairs are exist on ℓ_h for some $k' > k$. This means that the count of one of the end points of ℓ_h is k', which is also a contradiction. □

LEMMA A3. *Let ℓ_v denote a vertical line segment in r which has the same length as the height of r. Then the following statements are valid:*

— *there exist at most k max-regions that intersect ℓ_v, and*
— *there exist at most k up-stairs and k down-stairs that intersect ℓ_v.*

PROOF. We omit the proof, which follows easily from the proof of Lemma A2. □

Now the proof of Lemma 7 is given as follows:

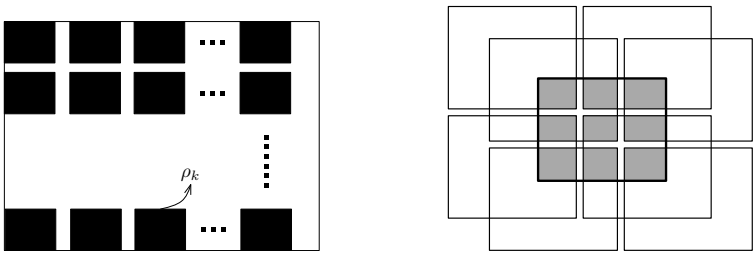
PROOF OF LEMMA 7. First, we can claim that there are at most k up-left stairs, k up-right stairs, k down-left stairs, and k down-right stairs in r since the corners of r should be covered by at most k stairs. By Lemma A2, there are at most k pairs of a left-stair and a right-stair on any horizontal line (including the top and bottom edges) in r, which implies that only k left/right-stairs among $2k$ left/right-stairs (i.e., k up-left/right stairs and k down-left/right stairs) should be laid on the same horizontal line. Furthermore, in order for each of these pairs to be a max-region, it should be associated with a pair of an up-stair and a down-stair. By Lemma A3, there are at most k pairs of an up-stair and a down-stair on any vertical line in r. Therefore, k^2 is the maximum number of max-regions that can reside in r. Intuitively, it can be seen that there are at most k horizontal layers each of which has at most k max-regions (see Figure A2(a)). □

Figure A2(b) shows how nine max-regions can reside in a rectangle when k is 3.

B. PROOF OF THEOREM 6

To prove Theorem 6, we use the following supporting lemma:

LEMMA B1. *Let ρ_x denote a max-region in $r \in R$, which is covered by x rectangles, and ℓ denote a horizontal or vertical line intersecting ρ_x, which is parallel to an edge of r and has the same length as the edge. Then there can exist at most x max-regions that intersect ℓ.*



(a) k max-regions on each line segment in a rectangle

(b) An example of k^2 max-regions in a rectangle when $k = 3$

Fig. A2. The illustration of Lemma 7

PROOF. First, it is obvious that Lemma A1 is valid in the weighted case. Thus, every max-region should have a pair of a left-stair and a right-stair as its side edges. In order to construct ρ_x, there should be at least $x + 1$ stairs (including a pair of parallel edges of r) on ℓ whose direction are towards ρ_x, as shown in Figure B1. Furthermore, except these $x + 1$ stairs, any other stairs on ℓ should not be towards ρ_x since ρ_x should be covered by only x rectangles (recall Property 2). This implies that we can construct only x pairs of a left-stair and a right-stair on ℓ by putting another stair right next to each of $x+1$ stairs that constitute ρ_x. By Lemma A1, there can be at most x max-regions on ℓ. □

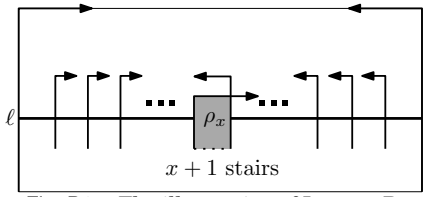


Fig. B1. The illustration of Lemma B1

Now the proof of Theorem 6 is given as follows:

PROOF OF THEOREM 6. We first prove that Lemma A2 and Lemma A3 are also valid in the weighted case when we use $\overline{k}$ instead of k. Since $\overline{k}$ is the maximum number of intersecting rectangles, any max-region is covered by at most $\overline{k}$ rectangles. This implies that, on any horizontal or vertical line segment passing through a max-region, there can exist at most $\overline{k}$ max-regions by Lemma B1, which derives Lemma A2 and Lemma A3. Therefore, Corollary 1 is also valid by substituting $\overline{k}$ for k, which says the upper bound of the number of max-regions should be less than $\overline{k}N$. □