Approximate MaxRS in Spatial Databases*

Yufei Tao[1,2] Xiaocheng Hu[1] Dong-Wan Choi[2] Chin-Wan Chung[2]

[1]Chinese University of Hong Kong
Hong Kong

[2]Korea Advanced Institute of Science and Technology
Korea

ABSTRACT

In the *maximizing range sum* (MaxRS) problem, given (i) a set P of 2D points each of which is associated with a positive weight, and (ii) a rectangle r of specific extents, we need to decide where to place r in order to maximize the *covered weight* of r – that is, the total weight of the data points covered by r. Algorithms solving the problem exactly entail expensive CPU or I/O cost. In practice, exact answers are often not compulsory in a MaxRS application, where slight imprecision can often be comfortably tolerated, provided that approximate answers can be computed considerably faster. Motivated by this, the present paper studies the $(1-\epsilon)$-*approximate MaxRS problem*, which admits the same inputs as MaxRS, but aims instead to return a rectangle whose covered weight is at least $(1-\epsilon)m^\star$, where $m^\star$ is the optimal covered weight, and ϵ can be an arbitrarily small constant between 0 and 1. We present fast algorithms that settle this problem with strong theoretical guarantees.

1. INTRODUCTION

Let P be a set of points in 2D space $\mathbb{R}^2$, where $\mathbb{R}$ represents the real domain. Each point $p \in P$ carries a positive value $w(p)$ as its *weight*. Given non-negative values a and b, the goal of the *maximizing range sum* (MaxRS) problem is to place an $a \times b$ rectangle r in $\mathbb{R}^2$ to maximize the *covered weight* of r, defined as:

$$covered\text{-}weight(r) = \sum_{p \in P \cap r} w(p). \quad (1)$$

In plain words, $covered\text{-}weight(r)$ equals the total weight of the points of P that are covered by r. As a special case, if every point in P has weight 1, then $covered\text{-}weight(r)$ simply indicates how many points of P fall in r.[1]

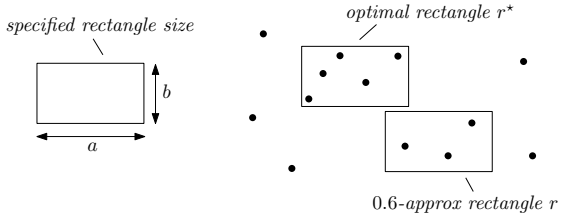


Figure 1: The MaxRS problem

Note that the position of r can be anywhere in the data space, namely, there are *infinitely* many possible rectangles that could have been chosen. To illustrate, consider that P is the set of black points in Figure 1, and (for simplicity) that all points have weight 1. Given the lengths of a and b as shown on the figure's left, an optimal solution to the MaxRS problem is the rectangle $r^\star$, which covers 5 points of P. One can easily verify that no rectangle of size $a \times b$ can enclose at least 6 points.

The MaxRS problem has been studied in both situations where P fits or does not fit in memory. Mentioning only the best results, in internal memory, Imai and Asano [6] solved the problem in $O(n \log n)$ time where $n = |P|$. Nandy and Bhattacharya [10] later gave another algorithm with the same cost. In external memory, Choi et al. [2] recently settled it in $O(SORT(n))$ I/Os, where $SORT(n)$ is the I/O cost of sorting n elements in the disk.

Motivation. All the solutions of [2, 6, 10] find an *optimal* rectangle, i.e., one that has the greatest covered weight. The motivation of this work is that, many applications of MaxRS can tolerate slight imprecision, if significant reduction in the computation time is possible. For example, a representative MaxRS application, as mentioned in [2], is to select a good location to open a new business, e.g., a pizza delivery store. In this scenario, each point corresponds to a residential location, and its weight gives the population at that location. The size of a rectangle r depends on the delivery range that the store is able to cover (the delivery range is better represented as a square, rather than a circle, because short distances on a road network are well approximated by Manhattan distances; this approach has been taken in [17]). The covered weight of r indicates the population inside the shop's delivery range. A manager would not mind retrieving a rectangle that covers, say, 1% less weight than an optimal one, especially if this will substantially reduce her/his waiting time.

*This work was supported in part by the WCU (World Class University) program under the National Research Foundation of Korea, and funded by the Ministry of Education, Science and Technology of Korea (Project No: R31-30007). Yufei Tao and Xiaocheng Hu were also supported in part by projects GRF 4166/10, 4165/11, and 4164/12 from HKRGC. Dong-Wan Choi and Chin-Wan Chung were also supported in part by the National Research Foundation of Korea grant funded by the Korean government (MSIP) (No. NRF-2009-0081365).

Proceedings of the VLDB Endowment, Vol. 6, No. 13

[1]Throughout the paper, given a rectangle r and a set P of points, we use $r \cap P$ to denote the set of points in P that are covered by r. This is a natural notation if one thinks of r as an (infinite) set of points.

MaxRS is also the key to discovering the highest *local data density*, an operation that finds applications in data mining. Specifically, given a square r of a "unit area", the local data density in r equals the number of points in r. Hence, the output of MaxRS corresponds to the square with the highest density. Once again, it would be useful if we could derive a very accurate estimate (with error less than 1%) of the highest density, by using significantly less time than finding the exact value.

Main Results. We study the $(1-\epsilon)$-*approximate* version of the MaxRS problem. Formally, given (i) a and b as defined before and (ii) an ϵ satisfying $0 < \epsilon < 1$, the goal of the $(1-\epsilon)$-*approximate MaxRS problem* is to report an $a \times b$ rectangle r whose covered weight is at least $(1-\epsilon)m^\star$, where $m^\star$ is the covered weight of an optimal rectangle $r^\star$. In other words, the *error* of r is at most $\epsilon m^\star$. We call r a $(1-\epsilon)$-*approximate answer*.

In Figure 1 (where all points have weight 1), for instance, the r as shown is a 0.6-approximate answer because it covers $3 = (1 - 0.4) \times 5$ points. Recall that $m^\star = 5$ is the number of points in an optimal answer $r^\star$. Note that here $\epsilon = 0.4$ is merely an example for illustration, while a useful ϵ in practice should be much smaller. To the best of our knowledge, $(1-\epsilon)$-approximate MaxRS is a new problem that has not been studied previously.

$(1-\epsilon)$-approximate MaxRS is useful only if it can be solved significantly faster than (exact) MaxRS. In the present paper, we prove that this is true. Specifically, we present an algorithm that returns a $(1-\epsilon)$-approximate answer with probability at least $1 - 1/n$ in only $O(n \log \frac{1}{\epsilon} + n \log\log n)$ time, *regardless of the data distribution*. Interestingly, when ϵ is fixed (i.e., ensuring the same precision guarantee that does not depend on n), the running time of our algorithm is $O(n \log\log n)$. Note that the factor $\log_2 \log_2 n$ grows very slowly with n, and is less than 5.4 even for $n = 2^{40}$. Our time complexity thus compares favorably with the $O(n \log n)$ time needed to find the exact answer: $\log_2 n = 40$ for $n = 2^{40}$.

Technical Overview. Our algorithm is surprisingly simple, but is fairly intuitively, and supported by solid theory. Let us start with a natural idea: why not just take a random sample set S of P, find the best rectangle on S, and then return this rectangle as an approximate answer for the original problem on P? A moment's thought, however, would reveal that this works only if $m^\star$ (i.e., the covered weight of an optimal rectangle) is large. Unfortunately, $m^\star$ can be arbitrarily small. In fact, when $m^\star$ drops below $1/\epsilon$, we must return an optimal answer if every weight is at least 1, because the permissible error is $\epsilon m^\star < 1$, namely, no error is allowed!

Our eventual success is harvested by refining this idea. In a nutshell, we divide the data space into a grid of cells, and calculate their densities (i.e., how much weight does a cell cover). Only certain "dense cells" are worth considering, whereas the other cells can be discarded outright. For each remaining cell, we apply the sampling approach as outlined earlier, and return the best rectangle found from all those cells. It turns out that, by concentrating on the dense cells, only a handful of samples are necessary to meet the precision constraint, which in turn leads to an excellent speed guarantee.

Implementing the above approach, however, gives rise to several challenges. First, the data space $\mathbb{R}^2$ has infinite extents on both dimensions, such that the number of cells is infinite. It is apparently unwise to enumerate all cells to obtain their densities. We overcome this obstacle by showing how to obtain the densities of all cells in $O(n)$ expected time, i.e., independently of the data space dimensions. Second, setting the density threshold to separate dense from sparse cells is crucial but non-trivial, as is a main technical contribution of this paper. The final challenge is to achieve a very high success probability of at least $1 - 1/n$. Somewhat unexpectedly, we prove that it suffices to repeat our algorithm only 4 times!

Experimentation Highlights. This paper contains an extensive experimental evaluation that demonstrates the excellent performance of the proposed techniques in both internal and external memory. Our algorithms always guaranteed results of extremely high quality (in fact, in many cases, they returned optimal answers), and were faster than the existing algorithms by a factor reaching *an order of magnitude*. This fully confirms the motivation of this work that we can significantly shorten the response time to users' requests with little sacrifice in quality. Furthermore, in our experiments, the proposed algorithms exhibited excellent scalability with the dataset size, particularly, at a much lower rate than the state of the art.

Paper Organization. In the next section, we clarify several fundamental properties of the MaxRS problem, and review the previous work related to ours. Section 3 elaborates the proposed $(1-\epsilon)$-approximate MaxRS algorithms, while Section 4 formally establishes their theoretical guarantees. Section 5 describes how to implement these algorithms I/O-efficiently in external memory. Section 6 presents our experimental results. Finally, Section 7 concludes the paper with a summary of our contributions.

2. PRELIMINARY

Next, we pave the way for our algorithmic discussion by elaborating on an inherent reduction commonly used to attack the MaxRS problem (Section 2.1), reviewing the existing algorithms for solving MaxRS exactly (Section 2.2), and clarifying the relevance and differences between our work and previous research in other related areas (Section 2.3).

To simplify discussion, let us assume (only in this section) that all points have unit weight 1. This allows us to illustrate the central ideas behind the algorithms without being distracted by the extra details for handling arbitrary weights.

2.1 Reduction to Equi-Rectangle Stabbing

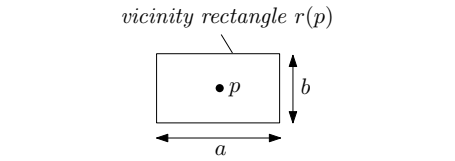
The MaxRS problem can be elegantly converted to an alternative problem called *equi-rectangle stabbing*. To explain, as before, let P, a and b be the inputs to the MaxRS problem, namely, P is a set of n points, and $a \times b$ specifies the size of the rectangle to be returned. Let us generate a set R of n rectangles, one from a distinct point in P, as follows. For each point $p \in P$, define a rectangle $r(p)$ as the $a \times b$ rectangle centered at p, as illustrated in Figure 2a. Then, R can be defined as:

$$R \quad = \quad \{r(p) \mid p \in P\}.$$

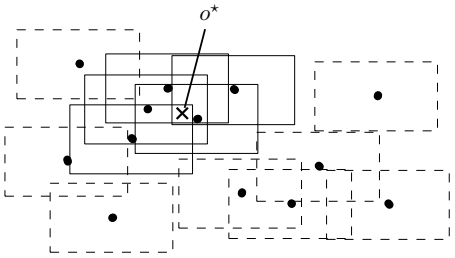
If we refer to $r(p)$ as the *vicinity rectangle* of p, then R includes the vicinity rectangles of all points $p \in P$. Figure 2b shows the rectangles of R converted from the set P of points in Figure 1 (R includes both the solid and dashed rectangles).

Given R, the goal of the *equi-rectangle stabbing* (ERS) *problem* is to find a point $o^\star$ that is covered by the largest number of rectangles in R, among all the possible points in the data space $\mathbb{R}^2$. The cross in Figure 2b demonstrates an optimal $o^\star$, which is covered by 5 rectangles in R (the solid ones). It is easy to verify that no point in the data space falls in 6 or more rectangles. Phrased differently, one can imagine *stabbing* the rectangles of R with a point, in which case the ERS problem aims at finding a point that can stab the most rectangles.

What makes the reduction work is the following fact: $r(p)$ covers a point $o \in \mathbb{R}^2$ *if and only if* p is covered by the $a \times b$ rectangle centered at o. In Figure 2b, this means that if we place an $a \times b$ rectangle $r^\star$ centered at $o^\star$, then by the fact that $o^\star$ falls in 5



(a) Vicinity rectangle



(b) R generated from P

Figure 2: Reduction from MaxRS to ERS

rectangles of R, we know that $r^\star$ covers 5 points of P. The $r^\star$ has to be an optimal answer to the MaxRS problem on P, judging from the optimality of $o^\star$ for the ERS problem on R.

The usefulness of the above reduction is that, it provides us with a new perspective to attack the MaxRS problem, by dealing with the ERS problem instead. Interestingly, this was exactly the approach taken by the state-of-the-art algorithms in both internal and external memory, as reviewed next.

2.2 Exact ERS Algorithms

This section will discuss the main ideas behind solving the ERS (and hence, MaxRS, by the reduction explained before) problem exactly. Not only does this shed light on the problem's characteristics, but also it will show that our techniques (presented in later sections) are based on drastically different rationales.

The basic methodology is *planesweep*. Imagine moving a horizontal line ℓ up from the initial position $y = -\infty$. In this process, ℓ starts intersecting a rectangle $r \in R$ when ℓ passes the bottom edge of r, and stops doing so when ℓ reaches the top edge of r. In between, r intersects ℓ into an interval on ℓ. At any moment, the intersection points between ℓ and the rectangles of R chop ℓ into disjoint intervals. Figure 3 shows the situation when ℓ passes the cross $o^\star$ in Figure 2b (omitting the rectangles that do not intersect ℓ). Notice that ℓ is divided into 10 intervals: AB, BC, ..., JK (ignoring the infinite intervals to the left and right of A and K, respectively).

Conceptually, the planesweep algorithm maintains a count for each interval on ℓ, which equals the number of rectangles in R covering the interval. In Figure 3, for instance, the count of AB is 1 because AB is covered by only 1 rectangle (i.e., the leftmost one). As more examples, the count of EF equals 5, while that of IJ equals 0. The algorithm keeps track of the interval with the highest count on the present ℓ, i.e., EF. The planesweep finishes when ℓ is above all the rectangles of R. Let v be the interval with the maximum count during the *entire* sweeping process. Then, any point of v can be returned as an optimal answer to the ERS problem. In our earlier example, one can verify that the count 5 of EF is the greatest during the whole process (on the dataset of Figure 2). Hence, the algorithm can return any point in EF as the final answer.

Implementation of the above idea is what distinguishes the internal memory algorithm [10] from the external memory ones [2, 4]. In memory, the intervals on ℓ can be maintained using a binary tree. Overall, the algorithm performs $2n$ updates (i.e., 2 per rectangle in

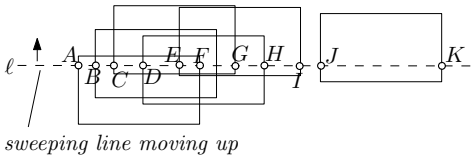


Figure 3: Solving the ERS problem: planesweep

R) on the binary tree, each of which takes $O(\log n)$ time, thus necessitating $O(n \log n)$ time overall [10]. As shown in [4], the same strategy also works in external memory, but entails $O(n \log_B n)$ I/Os (B is the size of a disk block), which unfortunately is very costly in practice. Choi et al. [2] remedied the problem with a different implementation which lowers the I/O cost substantially by a factor of nearly $\Omega(B)$.

2.3 Other Related Work

Facility Location. The ERS (a.k.a. MaxRS) problem can be regarded as an instance of *facility location optimization*, which is a broad class of problems that aim at finding an optimal location to place a "facility" in order to minimize/maximize a certain objective function. Specifically, in ERS, we want to find a point (i.e., facility) p to maximize the number (i.e., objective) of rectangles in R covering p.

Many concrete problems have been proposed in this class to suit the needs of various applications. A comprehensive survey is beyond the scope of this paper. At a high level, these problems can be divided into two categories: *monotonic* and *bichromatic*. In the former category, there is only one "type" of objects that can influence the choice of the best facility. The ERS problem belongs to this category. Other classic examples include the *smallest enclosing circle problem* [9], which returns the smallest circle that covers a set P of 2D points, and the *largest empty rectangle problem* [1], which returns the rectangle with the maximum area among all the rectangles that do not contain any point in P.

The bichromatic category, on the other hand, includes two sets: a set C of *consumers* and a set F of *facilities*. The goal is to find a new facility whose introduction serves the customers best, together with the existing facilities in F. Problems of this flavor [15, 16, 17, 18] have been investigated by the database community in the past few years. As a representative example, in the *min-dist optimal location problem* [17], C and F are sets of 2D points, such that the goal is to find a new point f in the data space to minimize the maximum distance between a customer $c \in C$ and her/his nearest facility in $F \cup \{f\}$.

This paper essentially tackles a form of *approximate facility location*, regarding which the closest previous work is due to Berg et al. [3]. Given a set P of 2D points, a radius d and a value $\epsilon \in (0, 1)$, they give an algorithm to return a circle with radius d that covers at least $(1 - \epsilon)m^\star$ points of P, where $m^\star$ is the maximum number of points that can be enclosed by a radius-d circle. Their algorithm runs in $O(n/\epsilon^3 + n \log n)$ time. Unfortunately, the factor $1/\epsilon^3$ is exceedingly large for reasonable value of ϵ in practice (e.g., at the order of 0.01) such that n/ϵ^3 overwhelms $n \log n$ for typical values of n ($\leq 2^{40}$), and is prohibitively expensive for a real system. Furthermore, the algorithm of [3], which is based on dynamic programming, is difficult to implement in external memory to attain even an I/O bound that comes close to $SORT(N)$, which as mentioned earlier is the cost of solving MaxRS exactly [2].

Range Aggregation. As nicely pointed out in [2], the MaxRS problem should not be confused with *range aggregation problem* [5, 8, 11, 12], where given a rectangle r and a set P of points in $\mathbb{R}^2$, we

want to return the number of points in P that are covered by r. Note that in this context the position of r is *known* as a problem input, as opposed to the need of *searching* for the best location to place a rectangle as in MaxRS. A naive solution to deal with MaxRS by way of range aggregation, is to collect all the possible rectangles in the data space, do a range-aggregation query for each one of them, and return the one covering most points. This solution is apparently infeasible even if the data space is discrete, because the number of possible rectangles is prohibitively large.

3. $(1-\epsilon)$-APPROXIMATE MAXRS

Now we formally define the $(1-\epsilon)$-*approximate MaxRS problem*. The input includes:

- A set P of 2D points, where each point $p \in P$ carries a positive weight $w(p)$.
- Non-negative real values a and b
- A real value ϵ such that $0 < \epsilon < 1$.

The objective is to return an $a \times b$ rectangle r in the data space $\mathbb{R}^2$ such that the covered weight of r – as defined in (1) – is at least $(1-\epsilon)m^\star$, where $m^\star$ is the covered weight of an optimal $a \times b$ rectangle $r^\star$ (i.e., $m^\star = \textit{covered-weight}(r^\star)$). Accordingly, we say that an $a \times b$ rectangle r is a $(1-\epsilon)$-*approximate answer* if

$$\textit{covered-weight}(r) \quad \geq \quad (1-\epsilon)m^\star$$

where $\epsilon m^\star$ is the *permissible error*.

We will use (a, b, P)-MaxRS to represent the corresponding *exact* MaxRS problem (that is, $r^\star$ is an optimal answer for (a, b, P)-MaxRS). We will also use (a, b, P)-count-MaxRS to denote the exact MaxRS problem where *all points in P are forced to have weight 1*. Clearly, every (a, b, P)-MaxRS problem has an (a, b, P)-count-MaxRS counterpart.

Next, we present our algorithms for the situation where P fits in memory. Our discussion will separate (deliberately) the algorithmic description from the theoretical analysis, which can be found in the next section. Finally, our techniques will be extended to external memory in Section 5.

3.1 Pitfall of Random Sampling

Let us first consider an intuitive algorithm as our first attempt to solve the $(1-\epsilon)$-approximate MaxRS problem:

Algorithm 1: SIMPLESAMPLING (s)

Input: Sample size s
Output: An $a \times b$ rectangle r

1 $S \leftarrow$ a random sample set of P with s points
2 $r \leftarrow$ an optimal answer to (a, b, S)-MaxRS (obtained by invoking the exact MaxRS algorithm of [10] on S)
3 **return** r

The algorithm runs in $O(s \log s)$ time as is determined by Line 2. Its rationale is that, since a random sample set S preserves the distribution of P, an optimal answer on S (i.e., rectangle r) might serve as a good approximate answer on P. In the experiments, we will see that this is the case when the permissible error $\epsilon m^\star$ is large. When $\epsilon m^\star$ is small, however, SIMPLESAMPLING becomes unreliable such that it is unable to achieve $(1-\epsilon)$-approximation unless s is exceedingly large.

We first give a quick explanation that is somewhat informal, but easy to understand. Consider that (i) all points have weight 1, and (ii) $\epsilon = 0.01$ but $m^\star < 100$ such that the permissible error $\epsilon m^\star < 1$. Therefore, a $(1-\epsilon)$-approximate answer is not allowed to have any error, and hence, will *have to be an optimal answer*. But how does SIMPLESAMPLING compute its answer r? From a random sample set S! In other words, the algorithm intends to find an optimal answer *without* even looking at all the points in P. This sounds too good to be true. Indeed, the consequence is that either the algorithm will fail to achieve the purpose, or S is such a large sample set that its size is already at the scale of P.

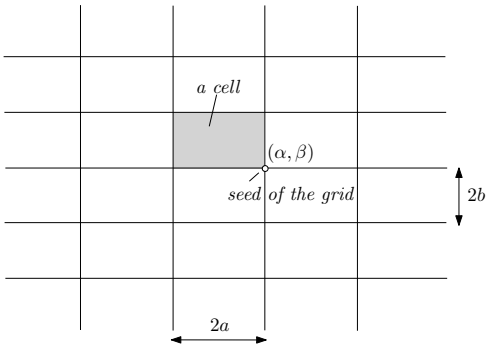


Figure 4: An (α, β)-grid

The next theorem establishes a strong negative result formalizing the above pitfall of SIMPLESAMPLING:

THEOREM 1. *Let δ be any arbitrarily small positive constant satisfying $0 < \delta < 1$. For any positive $\epsilon \leq 0.49$, there is a dataset on which, with probability at least $1-\delta$,* SIMPLESAMPLING *fails to return a $(1-\epsilon)$-approximate answer, unless $s = \Omega(n)$.*

PROOF. See appendix. □

Note that $s = \Omega(n)$ is not interesting because this means that the running time of SIMPLESAMPLING is $O(s \log s) = O(n \log n)$, i.e., asymptotically the same as solving the MaxRS *exactly*. Also, note that in reality a useful ϵ should be much lower than 0.49, implying that SIMPLESAMPLING will perform even worse than predicted by the above theorem. This negative result, therefore, essentially eliminates SIMPLESAMPLING as an adequate solution. Next, we will design a new algorithm to overcome the defects of SIMPLESAMPLING.

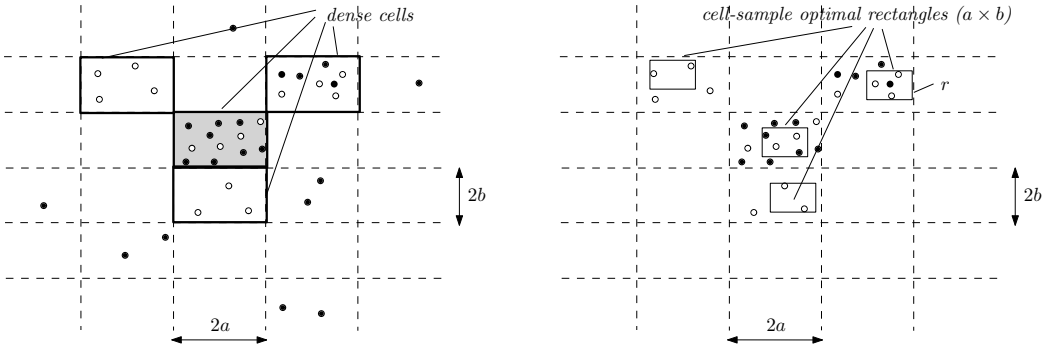
3.2 Grid Sampling

This section presents a new algorithm called GRIDSAMPLING for the $(1-\epsilon)$-approximate MaxRS problem. The algorithm incorporates two fresh ideas. First, when $m^\star$ is excessively low (i.e., even an optimal $a \times b$ rectangle covers little weight), we will solve the MaxRS problem *precisely*, i.e., obtaining an optimal answer directly. Second, opposite to the previous case, when $m^\star$ is not small, we will sample P only in some dense areas, as opposed to carrying out the sampling globally (i.e., on the entire P) as in SIMPLESAMPLING.

Grid and Cells. An obstacle in implementing the above strategy is that $m^\star$ is unknown. Fortunately, we do not need to know the exact value of $m^\star$, but instead, will be happy enough to derive an estimate that can be off by a factor of at most 4. For this purpose, we resort to a grid defined as:

DEFINITION 1 ((α, β)-GRID). *Given real values α, β, the* **(α, β)-grid** *is the set of vertical and horizontal lines defined respectively by:*

$$x = \alpha + i \cdot (2a),\ y = \beta + i \cdot (2b)$$



(a) Dense cells and samples (b) One optimal rectangle on the samples of each dense cell

Figure 5: GRIDSAMPLING algorithm ($s = 4$)

for all integers $i = -\infty, ..., -1, 0, 1, ..., \infty$, and a, b are parameters of the current MaxRS problem. Point (α, β) is called the **seed** *of the grid.*

Let G represent the (α, β)-grid as defined above. As shown in Figure 4, G is determined by an infinite set of vertical and horizontal lines, such that the distance between two consecutive vertical (horizontal) lines is $2a$ ($2b$), namely, *twice* as large as the parameter a (b) of the MaxRS problem at hand. We refer to the intersection of a pair of vertical and horizontal lines as a *junction*. Then, point (α, β) is one of the junctions.

We use the term *cell* to refer to a rectangle surrounded by two consecutive horizontal lines and two consecutive vertical lines in G (e.g., the gray area in Figure 4). Now, put P back to the data space, and view it together with G. Many cells of G are *empty* because they do not contain any point of P. Conversely, if a cell covers at least one point of P, it is *non-empty*. Given a cell c, we define its *weight*, denoted as $weight(c)$, as the total weight of the points in $c \cap P$. An empty cell thus has weight 0.

Proposed Algorithm. We are ready to explain our first algorithm, named GRIDSAMPLING, for solving the $(1-\epsilon)$-approximate MaxRS problem. First, we obtain an (α, β)-grid G by setting $\alpha = 0$ and $\beta = 0$. Then, we group the points of P by the cells they fall in. Let C be the set of *non-empty* cells of G.

Now, let us obtain the maximum weight t of the cells in G. Namely, $t = \max_{c \in C} weight(c)$. We use t to divide the non-empty cells into two types: (i) a cell $c \in C$ is *dense* if its weight is at least $t/4$, and (ii) *sparse*, otherwise. Figure 5a shows an example where P consists of all the points (black and white), and all points have weight 1. Here, $t = 12$, as is the weight of the gray cell. Then, a cell is dense if it encloses at least $t/4 = 3$ points. There are only 4 such cells as pointed out in the figure.

Let $s = O(\frac{1}{\epsilon^2}(\log \frac{1}{\epsilon} + \log n))$ be a parameter to GRIDSAMPLING. We proceed by *ignoring* all the sparse cells, i.e., the points therein will be removed from further consideration. For every dense cell c, define $S(c)$ as follows:

- If c covers at most s points, $S(c)$ includes all the points in c.
- Otherwise, $S(c)$ is a multi-set of size s obtained by sampling with replacement each point $p \in c \cap P$ with probability $w(p)/weight(c)$.

In any case, we refer to the points in $S(c)$ as *samples* of c. Next, for each dense cell c separately, we proceed as follows:

- If $S(c)$ already has all the points in c, run the exact MaxRS algorithm of [10] (reviewed in Section 2.2) to solve problem $(a, b, S(c))$-MaxRS.
- Otherwise, run the exact MaxRS algorithm of [10] to solve problem $(a, b, S(c))$-count-MaxRS, namely, *pretending that all points have weight 1*.[2]

In either case, let $r(c)$ be the rectangle returned by the algorithm of [10] for c. We call $r(c)$ a *cell-sample optimal rectangle*. Define:

DEFINITION 2 (QUALITY OF CELL-SAMPLE OPTIMAL). *If $r(c)$ is a cell-sample optimal rectangle from cell c, its* **quality** *equals the total weight of the points in $c \cap r(c)$.*

We return the cell-sample optimal rectangle with the highest quality as our final answer.

Algorithm 2: GRIDSAMPLING (G, s)

Input: (i) A grid G of Definition 1, and (ii) an integer s
Output: An $a \times b$ rectangle r

```
group the points of P by the cells of G they fall in
t ← the maximum weight of all cells
D ← the set of cells with weights ≥ t/4
q ← −∞ /* quality of the best cell-sample optimal */
for each cell c ∈ D do
    if S(c) = c ∩ P then
        r(c) ← an optimal answer to (a, b, S(c))-MaxRS (by using
        the exact algorithm of [10])
    else
        r(c) ← an optimal answer to (a, b, S(c))-count-MaxRS (by
        the algorithm of [10])
    if q < the quality of r(c) then
        r ← r(c) and q ← the quality of r(c)
return r
```

To illustrate, suppose that in Figure 5a the white points are the samples from the dense cells (assuming $s = 4$). Then, the exact algorithm is invoked four times in Figure 5a (because there are 4 dense cells), but each time on *only the white points of one cell*, as opposed to all the points in the cell. The figure also demonstrates the cell-sample optimal rectangles of the 4 dense cells. The quality of r equals 3, and is the highest among the 4 cell-sample optimal rectangles. We therefore return r as the answer. The pseudocode of GRIDSAMPLING is presented in Algorithm 2.

Cell Densities in Linear Time. At first glance, it may appear that obtaining the weights of all cells would be computationally expensive. This is not true. A crucial observation is that *there can be at*

[2]The reason solve $(a, b, S(c))$-count-MaxRS, instead of $(a, b, S(c))$-MaxRS, here is technically related to a concept called *ϵ-approximation*, as will be clarified in the proof of Lemma 5.

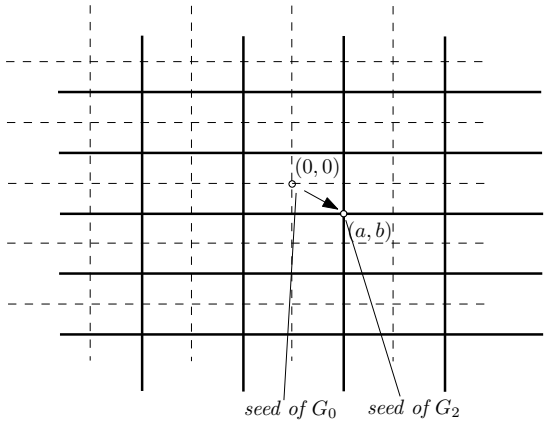


Figure 6: Shifting the (α, β)-grid

most $4n$ *non-empty cells*, because each point $p \in P$ can fall in at most 4 cells (when p is at a junction of the grid).

This observation enables us to compute the counts of all non-empty cells by applying hashing to implement Line 1 of Algorithm 2. Note that, for all the points $p = (x, y)$ in the same cell, their values of $\lceil \frac{x-\alpha}{2a} \rceil$ are all identical (the same is true for $\lceil \frac{y-\beta}{2b} \rceil$)[3]. If we treat the pair $(\lceil \frac{x-\alpha}{2a} \rceil, \lceil \frac{y-\beta}{2b} \rceil)$ as the ID of the cell containing p, Line 1 essentially groups the points of P by the IDs of their containing cells, as can be done by hashing in $O(n)$ expected time.

Weighted Sampling with Replacement. Line 6 requires a fast algorithm to obtain s weighted random samples with replacement. This problem has been well studied in the literature. In the appendix, we describe a method adopted in our implementation. The method completes the sampling in $O(n_c + s \log s)$ time, where $n_c = |c \cap P|$, i.e., the number of points in a cell c.

3.3 Grid Shifting

GRIDSAMPLING cannot guarantee returning a correct answer with a high probability. In this section, we show how to boost the success probability all the way to $1 - \frac{1}{n}$ (this is extremely close to 1 even for a moderately large n). In fact, all we need to do is to run GRIDSAMPLING 4 times with some carefully chosen grids, as explained below.

Recall that GRIDSAMPLING is invoked with the $(0, 0)$-grid G, which we denote by G_0 henceforth. Let us consider 3 more grids:

$$\begin{aligned} G_1 &: (a, 0)\text{-grid} \\ G_2 &: (a, b)\text{-grid} \\ G_3 &: (0, b)\text{-grid} \end{aligned}$$

where a, b are the parameters to the MaxRS problem at hand. Figure 6 illustrates the idea by presenting G_2 (solid grid), which is obtained by shifting G_0 (dashed grid) horizontally by distance a, and then, vertically by distance b.

Algorithm 3 presents a new algorithm called 4GRIDSAMPLING. All steps should be fairly clear except possibly Line 5. This step can be done with a single scan of P in $O(n)$ time.

This completes our description of the proposed algorithms for the $(1 - \epsilon)$-approximate MaxRS problem. As analyzed in the next section, 4GRIDSAMPLING runs in $O(n \log \frac{1}{\epsilon} + n \log \log n)$ time, and returns a $(1 - \epsilon)$-approximate answer with probability at least $1 - 1/n$.

[3]This assumes that p is not on the boundary of the cell. Otherwise, we assign p to at least two, but up to four, cells.

Algorithm 3: 4GRIDSAMPLING (s)

Input: An integer s
Output: An $a \times b$ rectangle r

1 $r_0 \leftarrow$ GRIDSAMPLING (G_0, s)
2 $r_1 \leftarrow$ GRIDSAMPLING (G_1, s)
3 $r_2 \leftarrow$ GRIDSAMPLING (G_2, s)
4 $r_3 \leftarrow$ GRIDSAMPLING (G_3, s)
5 **return** the rectangle among r_0, r_1, r_2, r_3 with the highest covered weight

4. THEORETICAL ANALYSIS

We now proceed to establish the theoretical guarantees of GRIDSAMPLING and 4GRIDSAMPLING developed in the previous section. Let us start with an easy lemma:

LEMMA 1. GRIDSAMPLING *and* 4GRIDSAMPLING *finish in* $O(n \log s)$ *expected time.*

PROOF. We will focus on GRIDSAMPLING because the cost of 4GRIDSAMPLING is at most 4 times higher. Line 1 of Algorithm 2 can be implemented in $O(n)$ expected time by hashing, as already explained in Section 3.2. For each dense cell c, we apply the algorithm of [10] on at most $\lambda_c = \min\{n_c, s\}$ points, where n_c is the number of points in $c \cap P$. Hence, we spend $O(\lambda_c \log \lambda_c + n_c)$ time per dense cell c (the time is dominated by the cost of sampling). The total time from Lines 2 to 9 is therefore bounded by

$$O\left(\sum_{\text{dense } c} (\lambda_c \log \lambda_c + n_c)\right) \quad = \quad O(n \log s)$$

where the equality used the facts that $\lambda_c \leq s$, and $\sum_{\text{dense } c} \lambda_c \leq \sum_{\text{dense } c} n_c \leq n$. □

Next, we focus on the quality of the answers returned by our algorithms. Henceforth, let $r^\star$ be an optimal $a \times b$ rectangle for the (a, b, P)-MaxRS problem at hand, namely, $m^\star = \textit{covered-weight}(r^\star)$. There is an interesting fact:

LEMMA 2. $r^\star$ *must be fully contained by a cell in one of* $G_0, G_1, ..., G_3$.

PROOF. Consider the cell of G_0 that covers the top-left corner of $r^\star$. Let $ABCD$ be the rectangle of the cell. Divide the rectangle into four $a \times b$ sub-rectangles, as shown in Figure 7, where they are labeled from 0 to 3.

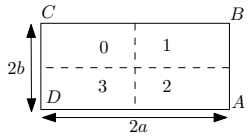


Figure 7: Proof of Lemma 2

It is easy to verify that if the top-left corner of $r^\star$ falls in rectangle $i \in [0, 3]$, then grid G_i has a cell that fully contains $r^\star$. □

Let $G^\star$ be the G_i (for some $i \in [0, 3]$) that has a cell $c^\star$ fully enclosing $r^\star$. The previous lemma shows that such a G_i always exists. The subsequent analysis focuses on the execution of GRIDSAMPLING on $G^\star$. We observe:

LEMMA 3. *When* GRIDSAMPLING *is executed on* $G^\star$*, it holds that* $t \in [m^\star, 4m^\star]$*, where* t *is the value obtained at Line 2.*

PROOF. Let c be the cell of $G^\star$ with the largest weight, i.e., $t = weight(c)$. The weight of $c^\star$ is at least $m^\star$ because $c^\star$ contains all points covered by $r^\star$. Note that c may or may not be $c^\star$. In either case, due to the definition of c, we know that $t \geq weight(c^\star) = m^\star$.

To prove $t \leq 4m^\star$, notice that c can be divided into four $a \times b$ rectangles, none of which can have a covered weight greater than $r^\star$ (otherwise, we have found a rectangle with a higher covered weight, contradicting the definition of $r^\star$). □

COROLLARY 1. *When* GRIDSAMPLING *is executed on $G^\star$, $c^\star$ must be dense.*

PROOF. This cell must have weight at least $m^\star$, which is at least $t/4$ by Lemma 3. □

The following analysis is divided into two parts, depending on the comparison between $n_{c^\star}$ and s, where $n_{c^\star}$ is the number of points in $P \cap c^\star$.

Case 1: $n_{c^\star} \leq s$. We have:

LEMMA 4. *When executed on $G^\star$ and $n_{c^\star} \leq s$,* GRIDSAMPLING *returns an optimal (i.e., exact) answer.*

PROOF. When $n_{c^\star} \leq s$, Line 7 of Algorithm 2 will discover $r^\star$ when processing $c^\star$ (which must be dense by Corollary 1) because $S(c^\star)$ includes *all* the points of $P \cap c^\star$. □

Case 2: $n_{c^\star} > s$. We say that a rectangle is *bad* if its covered weight is less than $(1-\epsilon)m^\star$; otherwise, the rectangle is *good*. No bad rectangle can be returned as a $(1-\epsilon)$-approximate answer, while conversely, *any* good rectangle is a $(1-\epsilon)$-approximate answer. This observation is useful for proving the next imperative lemma:

LEMMA 5. *When executed on $G^\star$ and $n_{c^\star} > s$,* GRIDSAMPLING *returns a $(1-\epsilon)$-approximate answer with probability at least $1 - \frac{1}{n}$.*

PROOF. When processing $c^\star$ (which must be dense by Corollary 1), Line 8 of GRIDSAMPLING finds a cell-sample optimal rectangle $r(c^\star)$. We will prove that, with probability at least $1 - \frac{1}{n}$, the following statements hold simultaneously:

- $\boldsymbol{P_1}$: $r(c^\star)$ is good.
- $\boldsymbol{P_2}$: the quality of $r(c^\star)$ (see Definition 2) is at least $(1-\epsilon)m^\star$.

Once this is done, we complete the proof of the lemma with the following argument. Suppose that, for any dense cell c', the cell-sample optimal rectangle $r(c')$ is bad. Then, the quality of $r(c')$ cannot be higher than the total weight of the points in $P \cap r(c')$, which in turn is less than $(1-\epsilon)m^\star$ by the definition of bad rectangle. Therefore, $r(c')$ cannot be returned by GRIDSAMPLING, implying that GRIDSAMPLING can return only a good rectangle.

Next, we establish $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ using the concept of ϵ-*approximation* [13]. Let $P(c^\star)$ be the set of points in $P \cap c^\star$. Given a rectangle r, we denote by $weight(r, c^\star)$ the total weight of the points of P that fall in $c^\star \cap r$. Note that, for any bad rectangle r_{bad}, it holds that $weight(r_{bad}, c^\star) \leq weight(r_{bad}) < (1-\epsilon)m^\star$. On the other hand, for $r^\star$, we know $weight(r^\star, c^\star) = weight(r^\star) = m^\star$.

At Line 6, GRIDSAMPLING obtains a weighted random sample $S(c^\star)$ of $P(c^\star)$. The size of $S(c^\star)$ is s. As proved in [13], by choosing $s = \frac{\rho}{(\epsilon/8)^2}(\log\frac{1}{\epsilon/8} + \log n)$ for a suitable constant ρ, with probability at least $1 - \frac{1}{n}$, $S(c^\star)$ is an $(\epsilon/8)$-*approximation* of $P(c^\star)$. This means that for any rectangle r, it holds that

$$\left| \frac{|S(c^\star) \cap r|}{|S(c^\star)|} - \frac{weight(r, c^\star)}{weight(c^\star)} \right| \leq \epsilon/8. \quad (2)$$

By setting r to $r^\star$ in the above, we obtain:

$$\begin{aligned} \frac{|S(c^\star) \cap r^\star|}{|S(c^\star)|} &\geq \frac{weight(r^\star, c^\star)}{weight(c^\star)} - \epsilon/8 \\ &= \frac{m^\star}{weight(c^\star)} - \epsilon/8. \end{aligned} \quad (3)$$

On the other hand, by setting r to any bad rectangle r_{bad} in Inequality 2, we have:

$$\begin{aligned} \frac{|S(c^\star) \cap r_{bad}|}{|S(c^\star)|} &\leq \frac{weight(r_{bad}, c^\star)}{weight(c^\star)} + \epsilon/8 \\ &< \frac{(1-\epsilon)m^\star}{weight(c^\star)} + \epsilon/8. \end{aligned} \quad (4)$$

Lemma 3 tells us that $weight(c^\star) \in [m^\star, 4m^\star]$. Under this condition, it holds that

$$\frac{(1-\epsilon)m^\star}{weight(c^\star)} + \epsilon/8 \leq \frac{m^\star}{weight(c^\star)} - \epsilon/8 \quad (5)$$

To see this, note that the above is equivalent to:

$$\begin{aligned} \frac{(1-\epsilon)m^\star}{weight(c^\star)} + \epsilon/4 &\leq \frac{m^\star}{weight(c^\star)} \quad \Leftrightarrow \\ \epsilon/4 &\leq \frac{\epsilon m^\star}{weight(c^\star)} \quad \Leftrightarrow \\ weight(c^\star) &\leq 4m^\star \end{aligned}$$

which is true. Thus, by combining (3), (4), and (5), we obtain:

$$\begin{aligned} \frac{|S(c^\star) \cap r^\star|}{|S(c^\star)|} &> \frac{|S(c^\star) \cap r_{bad}|}{|S(c^\star)|} \quad \Rightarrow \\ |S(c^\star) \cap r^\star| &> |S(c^\star) \cap r_{bad}|. \end{aligned}$$

This means that the cell-sample optimal rectangle $r(c^\star)$ chosen for $c^\star$ by Line 9 of GRIDSAMPLING cannot be a bad rectangle, because $r^\star$ covers more points in $S(c^\star)$ than *any* bad rectangle[4]. It thus follows that Statement $\boldsymbol{P_1}$ holds.

It remains to prove Statement $\boldsymbol{P_2}$. By definition of $r(c^\star)$, we know $|S(c^\star) \cap r(c^\star)| \geq |S(c^\star) \cap r^\star|$. Setting r to $r(c^\star)$ in (2) gives:

$$\frac{|S(c^\star) \cap r(c^\star)|}{|S(c^\star)|} \leq \frac{weight(r(c^\star), c^\star)}{weight(c^\star)} + \epsilon/8 \quad (6)$$

Combining the above with (3) and the fact $|S(c^\star) \cap r(c^\star)| \geq |S(c^\star) \cap r^\star|$ leads to:

$$\begin{aligned} \frac{m^\star}{weight(c^\star)} - \epsilon/8 &\leq \frac{weight(r(c^\star), c^\star)}{weight(c^\star)} + \epsilon/8 \quad \Rightarrow \\ weight(r(c^\star), c^\star) &\geq m^\star - (\epsilon/4) \cdot weight(c^\star) \\ &\geq m^\star - (\epsilon/4) \cdot 4m^\star \\ &= (1-\epsilon)m^\star. \end{aligned}$$

Note that $weight(r(c^\star), c^\star)$ is exactly the quality of $r(c^\star)$. We thus have completed the whole proof. □

[4]This is also why Line 9 of GRIDSAMPLING solves problem $(a, b, S(c))$-count-MaxRS, instead of $(a, b, S(c))$-MaxRS.

Putting everything together, we arrive at the main result of this paper:

THEOREM 2. 4GRIDSAMPLING *terminates in* $O(n \log \frac{1}{\epsilon} + n \log \log n)$ *expected time, and returns a* $(1-\epsilon)$*-approximate answer with probability at least* $1 - \frac{1}{n}$.

PROOF. The computation time follows directly from Lemma 1 and the fact that $s = O(\frac{1}{\epsilon^2}(\log \frac{1}{\epsilon} + \log n))$. As mentioned before, Lemma 2 proves the existence of $G^\star$. Consider now the execution of GRIDSAMPLING on $G^\star$. If $n_{c^\star} \leq s$, Lemma 4 proves that GRIDSAMPLING returns an optimal (i.e., precise) answer. Otherwise, Lemma 5 proves that the algorithm returns a $(1-\epsilon)$-answer with probability at least $1 - \frac{1}{n}$. □

5. EXTENSION TO EXTERNAL MEMORY

In this section, we adapt the proposed algorithms to the scenario where P does not fit in memory, but needs to be stored in the disk.

SIMPLESAMPLING. The extension of this algorithm to external memory is straightforward. One can implement Line 1 of Algorithm 1 using any existing disk-based sampling method (see [7]). Then, Line 2 can be carried out either in memory if the sample set is small, or by the I/O-efficient MaxRS algorithm of [2] otherwise.

GRIDSAMPLING and 4GRIDSAMPLING. We will concentrate on GRIDSAMPLING because 4GRIDSAMPLING is a simple extension. A major change in GRIDSAMPLING is the implementation of Line 1 in Algorithm 2. Recall that, in internal memory, this was taken care of by hashing in $O(n)$ time. The same strategy, however, is not suitable for external memory where hashing requires $O(n)$ I/Os, which is exceedingly expensive as it amounts to scanning the dataset $\Omega(B)$ times (B is the block size)! The deficiency is due to the well-known fact that, in general, hashing benefits little from *blocking* by necessitating highly random traffic that exhibits little disk locality [14].

Let G be the grid that GRIDSAMPLING runs with. Remember that Line 1 essentially performs a group-by of P (by the ids of the cells in G the points fall in, as mentioned in Section 3.2). Group-by is such a fundamental operator in database systems that it can be expeditiously performed by sorting. This is exactly the solution adopted in our implementation.

Once the points have been grouped by the cells, the remainder of the algorithm breezes through with just another sequential scan of P. Specifically, we process each group as follows. Let c be the cell of G that the group corresponds to. We take a weighted random sample set $S(c)$ from the points in the group. $S(c)$, which contains at most s points, is small enough to fit in memory, which enables us to find $r(c)$ (at Line 7 or 9 of Algorithm 2) without any I/O accesses.

In terms of I/O efficiency, the bottleneck of GRIDSAMPLING is the group-by operation which as mentioned earlier requires sorting. There are, however, scenarios where sorting is not required, such that the group-by can be accomplished in $O(n/B)$ I/Os. One such scenario is when the number h of non-empty cells in G does not exceed the number of blocks in memory minus one. In such a scenario, we can allocate one block of memory as the input buffer to read the input P, and meanwhile, use a memory block for each non-empty cell c as the output buffer for writing to the disk the points falling in c. In this way, the group-by finishes by reading and writing the dataset exactly once, respectively, so that GRIDSAMPLING requires only linear I/Os overall.

Another, more effective, heuristic leverages the fact that, the sample size s of each cell c (recall that s is a parameter to GRIDSAMPLING) is small – at the order of $\log_2 n$ in practice. When $hs \leq M$ with M being how many points can fit in memory, we can keep the sample sets of all non-empty cells memory-resident. In such a case, GRIDSAMPLING terminates by reading P only once (without any write I/Os) because, after group-by, GRIDSAMPLING works only on the sample sets only.

6. EXPERIMENTS

In this section, we present an empirical evaluation of the proposed techniques for solving the $(1-\epsilon)$-approximate MaxRS problem. First, Section 6.1 clarifies the setup of our experiments. Then, Section 6.2 (6.3) demonstrates the results in internal (external) memory.

6.1 Setup

Datasets. Our experimentation was based on 5 real datasets named CAR, RAIL, UAC, BLOCK, and NE, which contained 2.2m, 3.4m, 8.0m, 11.5m, and 20.0m points, respectively. The data space has a length of 10^9 on each dimension. CAR and RAIL were generated from transportation networks. Specifically, each point in CAR (RAIL) represents a junction in the road (railway) system of California (the entire US). On the other hand, the points of UAC, BLOCK, and NE corresponded to actual geographic locations in urban areas, city blocks, and the northeastern of the US, respectively. These datasets are the product of the TIGER project at the US Census Bureau, and have been frequently utilized in the experiments of previous work in spatial databases. For every data point, we generated its weight uniformly in the range from 1 to 50.

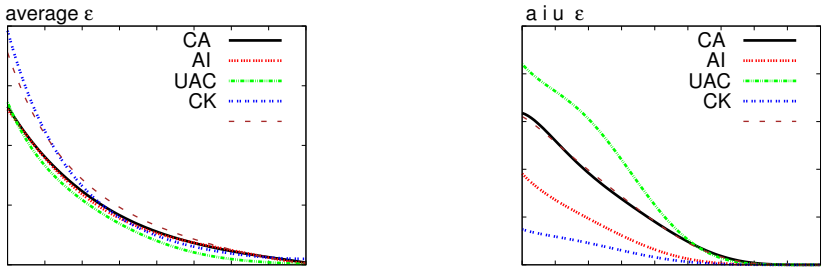
Machine and OS. All the experiments were performed on a machine running an Intel 3GHz CPU with 8 GB of memory. The operating system was Linux. For I/O-related experiments, the block size was fixed to 4k bytes.

Competing Methods. In internal memory, we compared the proposed algorithm 4GRIDSAMPLING (Algorithm 3) against SIMPLESAMPLING (Algorithm 1), and the algorithm of [10], denoted as OPTIMAL, for solving the MaxRS problem exactly. Similarly, in external memory, the comparison was among 4GRIDSAMPLING, SIMPLESAMPLING (implemented as described in Section 5), and the exact algorithm of [2], also denoted as OPTIMAL when no ambiguity can arise. We did not include GRIDSAMPLING because it is merely an intermediate product in the development of 4GRIDSAMPLING.

Parameters. The $(1-\epsilon)$-approximate MaxRS problem is specified by parameters ϵ, a and b. We eliminated b by equating it to a, namely, the requested rectangle was always a square. However, a itself is hardly a characteristic factor determining the behavior of approximation algorithms, because it is overly sensitive to the scaling of the data space's dimensions and the data distribution. What is directly influential is $m^\star$, namely, the total weight of the points covered by an optimal rectangle. Intuitively, the approximation problem is difficult when $m^\star$ is low because the permissible error is small.

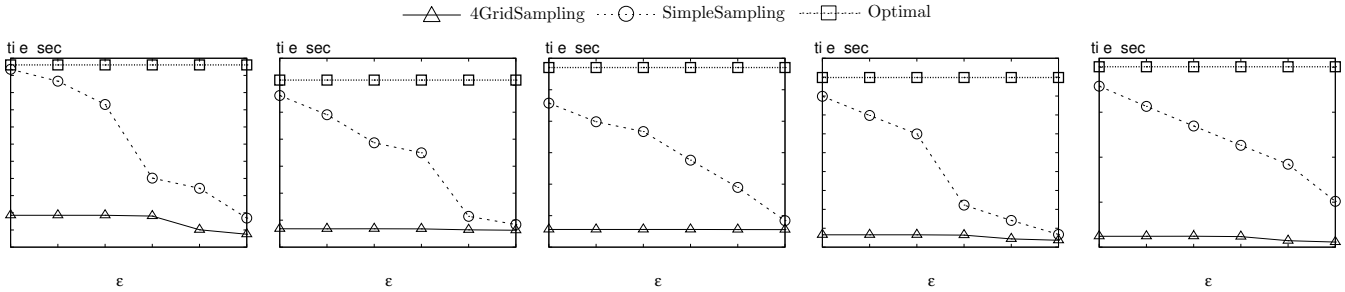
We varied ϵ among 0.25%, 0.5%, <u>1%</u>, 2%, 4%, and 8%, while $m^\star$ was changed among 256, 512, 1024, 2048, 4096, 8192 and <u>16384</u>, where the underlined were the default values. Once $m^\star$ was decided, so was a, which was set to the minimum value ensuring that an optimal $a \times a$ square should enclose at least weight $m^\star$.

Workload and Measurement. 4GRIDSAMPLING and SIMPLESAMPLING are randomized algorithms. Hence, their performance is not necessarily the same each time. Because of this, each value reported in our diagrams for either algorithm was obtained from a



(a) SIMPLESAMPLING (b) 4GRIDSAMPLING

Figure 8: Effect of sample size on accuracy



(a) CAR (b) RAIL (c) UAC (d) BLOCK (e) NE

Figure 9: Running time vs. accuracy

workload, in which we ran the algorithm on the same problem 20 times. Its efficiency was measured as the average CPU or I/O time of all its runs in a workload.

For 4GRIDSAMPLING, we required that it should output a correct $(1-\epsilon)$-approximate answer in *all* those runs, i.e., it *never* failed. We were able to do so because, as proven in Theorem 2, this algorithm succeeds with extremely high probability. SIMPLESAMPLING, however, does not even come close to having such a property, such that we relaxed the precision requirement by asking it to achieve an error of $\epsilon m^\star$ only *on average*. In other words, it was acceptable even if SIMPLESAMPLING failed to guarantee $(1-\epsilon)$-approximation in some runs. In this way, we essentially applied *a double standard in favor of* SIMPLESAMPLING.

6.2 Internal Memory and Result Quality

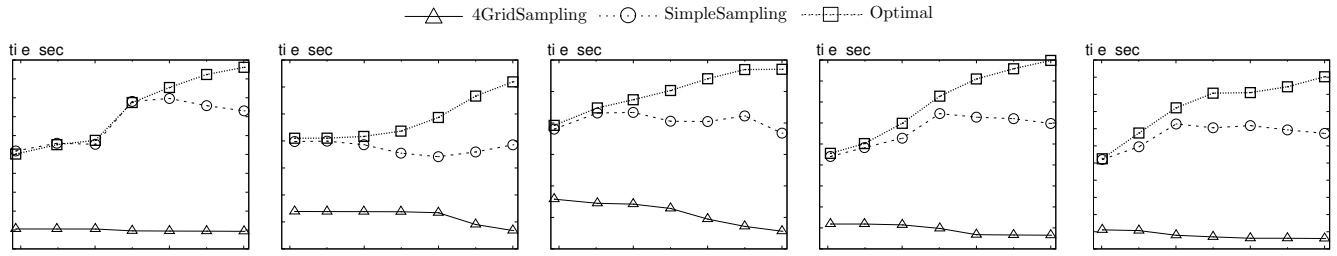
Pitfall of SIMPLESAMPLING. The first experiment aimed at revealing how large a sample set is demanded by SIMPLESAMPLING in order to return accurate answers. We pointed out in Section 3.1 the unreliability of SIMPLESAMPLING by explaining why its sample size s has to be exceedingly large to guarantee good approximation. Figure 8a shows that this is indeed true in reality. For each dataset, we deployed SIMPLESAMPLING on the MaxRS problem with $m^\star$ set to its default value 16384, by increasing s from $0.1n$ all the way to $0.9n$. In the meantime, we measured the ϵ that the algorithm was able to achieve (in the average sense, as mentioned in Section 6.1). Observe from Figure 8a that a descent amount of samples is needed to strike good precision. In particular, to cater for $\epsilon = 0.01$, s needed to be at least $0.5n$ in all cases, i.e., half of the dataset must be sampled!

Sample Size of 4GRIDSAMPLING. Remember that 4GRIDSAMPLING also admits a sample size s as a parameter. The proof of Theorem 2 showed that a good theoretical value for s is $O(\frac{1}{\epsilon^2}\log n)$. Next, we demonstrate good values for s in practice. For each dataset, fixing $m^\star$ to its default, we changed s from $10\log_2 n$ to $100\log_2 n$, while monitoring the ϵ guaranteed by the algorithm (in the *strict* sense – remember that 4GRIDSAMPLING was never allowed to fail). The results are illustrated in Figure 8b. Notice that the error of 4GRIDSAMPLING decreased dramatically as soon as s started to grow, such that by the time s reached $100\log_2 n$, its ϵ had become 0, indicating that 4GRIDSAMPLING always found an optimal answer. We recommend $s = \frac{1}{100\epsilon^2}\log_2 n$ for practical use, which worked very well in all our experiments.

Efficiency. We now proceed to assess the execution time of alternative algorithms. Fixing $m^\star$ to 16384, the next experiment deployed 4GRIDSAMPLING and SIMPLESAMPLING to find $(1-\epsilon)$-approximate answers (in the strict and average senses, respectively; this is the same in all experiments below) for each dataset, when ϵ increased from 0.25% to 8%. Figure 9 compares the running time of the two algorithms against OPTIMAL. Evidently, 4GRIDSAMPLING was significantly faster than OPTIMAL by an order of magnitude in all datasets. This confirms the very original motivation of studying approximate MaxRS, i.e., boosting the efficiency considerably by sacrificing little precision. SIMPLESAMPLING was also much slower than 4GRIDSAMPLING, except when the permissible error was large enough, so that SIMPLESAMPLING could be run on a small sample set.

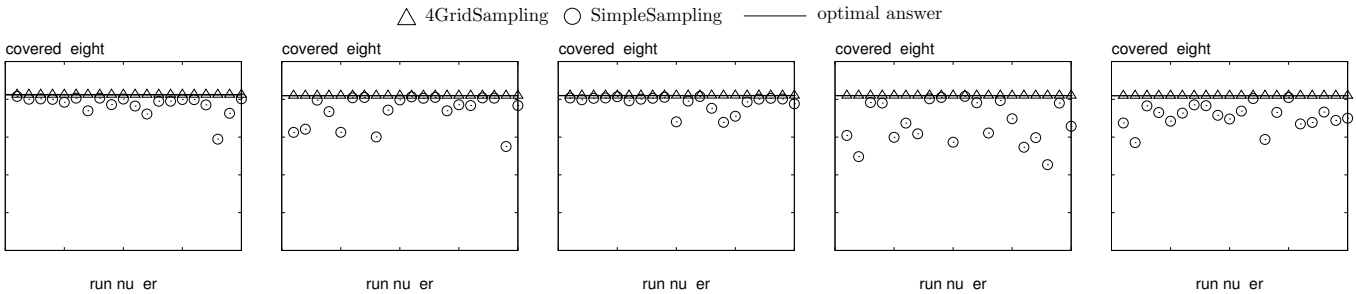
To investigate the influence of $m^\star$, we set ϵ to its default value 1%, varied $m^\star$ in its spectrum, and gauged the running time of 4GRIDSAMPLING, SIMPLESAMPLING and OPTIMAL. The result is presented in Figure 10. 4GRIDSAMPLING consistently outperformed all its competitors by a comfortable margin in all cases.

Error Distribution. The next experiment was designed to answer the following question: *how good are the answers of* SIMPLESAMPLING *if it uses the same amount of time as* 4GRIDSAMPLING*?* Remember that the running time of SIMPLESAMPLING depends solely on the number s of samples used. Therefore, it is possible



(a) CAR (b) RAIL (c) UAC (d) BLOCK (e) NE

Figure 10: Running time vs. the number of points in an optimal rectangle



(a) CAR (b) RAIL (c) UAC (d) BLOCK (e) NE

Figure 11: Comparison of error distributions

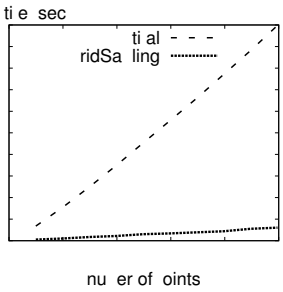


Figure 12: Scalability with dataset size

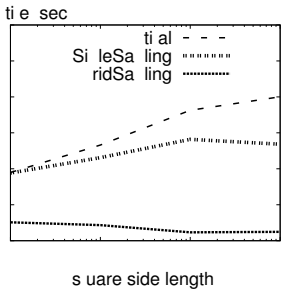


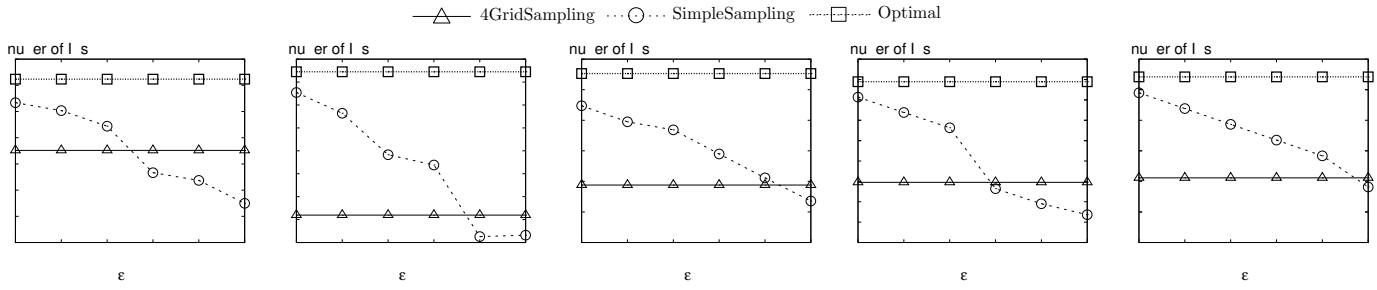
Figure 13: Running time vs. query rectangle size

to accurately adjust its running time by controlling s. Setting both $m^\star$ and ϵ to their default values, we first executed 4GRIDSAMPLING on a dataset. Then, given an identical (query) rectangle size, SIMPLESAMPLING was parameterized to run on the same dataset for as long as 4GRIDSAMPLING did. This process was repeated 20 times.

For every dataset, Figure 11 gives the output quality of all the 20 runs of 4GRIDSAMPLING and SIMPLESAMPLING, respectively. For each run, we show the covered weight of the rectangle returned by each algorithm. In each diagram, the precise answer is given as a horizontal line. In other words, the closer an approximate answer is to the line, the better approximation it offers. As expected, all the results of 4GRIDSAMPLING were extremely accurate – in most cases, the algorithm actually found the optimal answers! In contrast, the results of SIMPLESAMPLING were much worse. In particular, when $m^\star$ is fixed, SIMPLESAMPLING tends to incur higher error in a larger dataset (observe that its errors were especially significant in BLOCK and NE). These results thus further confirm the superiority of 4GRIDSAMPLING over SIMPLESAMPLING.

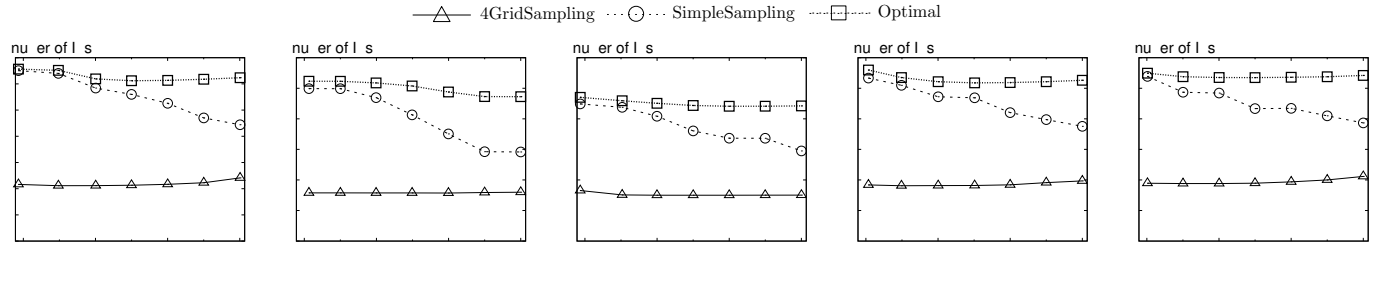
Scalability. We also studied how the cost of 4GRIDSAMPLING scales with the dataset cardinality n. For this purpose, we took the largest dataset NE, and created its miniatures of various sizes by random sampling (e.g., a miniature of 2m was obtained by sampling 10% of NE). Using the defaults of ϵ and $m^\star$, Figure 12 plots the running time of 4GRIDSAMPLING and OPTIMAL when they were executed on each of the miniatures. It is clear that 4GRIDSAMPLING scaled much better than OPTIMAL. This is expected because 4GRIDSAMPLING has a complexity of $O(n \log \log n)$ (ignoring the part of its complexity related to ϵ, which was fixed here), as opposed to the $O(n \log n)$ time demanded by OPTIMAL.

Influence of Rectangle Size. Recall that, in all our experiments, the target rectangle was an $a \times a$ square. As mentioned earlier, the value of a is not a characteristic factor on query efficiency. Nevertheless, in practice, it is natural for a user to specify a directly, since the value of $m^\star$ is difficult to predict. Therefore, the last set of experiments in this section aims to give the reader some sense of how the algorithms behave as this parameter changes. Again using dataset NE and fixing ϵ to 1%, we measured the running time of 4GRIDSAMPLING, SIMPLESAMPLING, and OPTIMAL as a increased from 10^3 to 10^6 (recall that the data space has an axis length of 10^9). The results are demonstrated in Figure 13, where the behavior of all methods is similar to that in Figure 10e.



(a) CAR (b) RAIL (c) UAC (d) BLOCK (e) NE

Figure 14: I/O cost vs. accuracy



(a) CAR (b) RAIL (c) UAC (d) BLOCK (e) NE

Figure 15: I/O cost vs. optimal answer

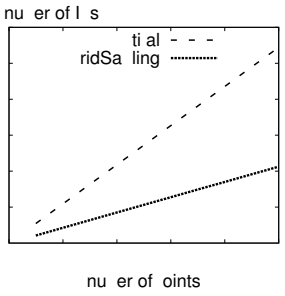


Figure 16: I/O scalability with dataset size

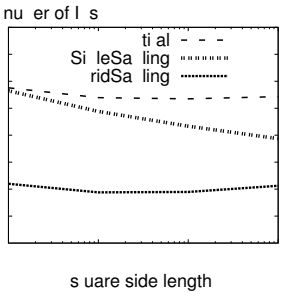


Figure 17: I/O cost vs. query rectangle size

6.3 External Memory

We now turn our attention to the scenario where the dataset does not fit in memory, thus rendering the I/O overhead the dominant cost. The amount of memory allocated to each algorithm was 10% of the number of blocks occupied by the underlying dataset. Result quality is no longer a concern of the subsequent experiments because each algorithm returned the same answers as in the internal-memory cases, whose accuracy has already been examined before.

Fixing $m^\star$ to 16384, Figure 14 compares the I/O cost of 4GRIDSAMPLING, SIMPLESAMPLING and OPTIMAL on each dataset, when ϵ grows from 0.25% to 8%. Conversely, setting ϵ to 1%, Figure 15 presents the comparison as a function of $m^\star$. The general observations are analogous to those that we made earlier from Figures 9 and 10. In particular, 4GRIDSAMPLING was faster than OPTIMAL by a factor between 2 and 3, and also outperformed SIMPLESAMPLING significantly except when ϵ was large.

We evaluated the I/O scalability of 4GRIDSAMPLING by repeating the experiment of Figure 16 in external memory. Both 4GRIDSAMPLING and OPTIMAL scaled gracefully with the dataset cardinality, with 4GRIDSAMPLING being the clear winner. Finally, we also repeated the experiment of Figure 13 in the I/O context, with the results given in Figure 17.

7. CONCLUSIONS

MaxRS is an interesting problem in spatial databases that has caught the attention of the database community only recently. Although algorithms for solving the problem exactly are already available, they unfortunately incur significant CPU or I/O time, when the input set P can or cannot be accommodated in memory, respectively. As a result, those algorithms fall short in fulfilling the need of a real system in providing the users with speedy answers.

This paper offers a remedy by trading away slight precision for significant improvement in efficiency. We propose a new problem called $(1-\epsilon)$-*approximate MaxRS* that returns a solution that can be worse than an optimal solution by a factor of at most ϵ, where ϵ can be an arbitrarily small positive constant satisfying $0 < \epsilon < 1$. We present algorithms for settling this problem with a fraction of the cost of the state-of-the-art approaches. Our techniques are based on innovative ideas that are drastically different from those of the previous work, and hence, are of independent interest.

8. REFERENCES

[1] B. Chazelle, R. L. S. D. III, and D. T. Lee. Computing the largest empty rectangle. *SIAM J. of Comp.*, 15(1):300–315, 1986.

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

[3] M. de Berg, S. Cabello, and S. Har-Peled. Covering many or few points with unit disks. *Theory Comput. Syst.*, 45(3):446–469, 2009.

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

[5] S. Govindarajan, P. K. Agarwal, and L. Arge. CRB-tree: An efficient indexing scheme for range-aggregate queries. In *ICDT*, pages 143–157, 2003.

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

[7] C. Jermaine, A. Pol, and S. Arumugam. Online maintenance of very large random samples. In *SIGMOD*, pages 299–310, 2004.

[8] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *SIGMOD*, pages 401–412, 2001.

[9] N. Megiddo. Linear-time algorithms for linear programming in R^3 and related problems. *SIAM J. of Comp.*, 12(4):759–776, 1983.

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

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

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

[13] V. Vapnik and A. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16(2):264–280, 1971.

[14] Z. Wei, K. Yi, and Q. Zhang. Dynamic external hashing: the limit of buffering. In *SPAA*, pages 253–259, 2009.

[15] R. C.-W. Wong, M. T. Ozsu, P. S. Yu, A. W.-C. Fu, and L. Liu. Efficient method for maximizing bichromatic reverse nearest neighbor. *PVLDB*, 2(1):1126–1137, 2009.

[16] X. Xiao, B. Yao, and F. Li. Optimal location queries in road network databases. In *ICDE*, pages 804–815, 2011.

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

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

Proof of Theorem 1

It suffices to consider the case where all points have weight 1. Set $\epsilon = 0.49$. We will construct an instance of the $(1-\epsilon)$-approximate MaxRS problem with $m^\star = 2$ such that the permissible error is $\epsilon m^\star = 0.49 \times 2 < 1$. In other words, a $(1-\epsilon)$-approximate answer has to be an optimal answer.

Specifically, P is designed as shown in Figure 18. Among its n points, $n-2$ of them are placed evenly along a circle; they are called *border points*. Two points are placed very close to each other at the circle's center; they are called *center points*, which can be covered by an $a \times b$ rectangle. The radius of the circle is sufficiently large so that, if an $a \times b$ rectangle covers a border point, it cannot contain any other point.

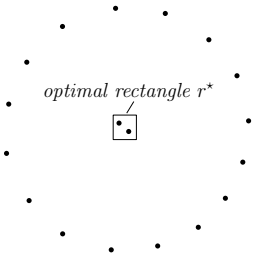


Figure 18: Bad input for SIMPLESAMPLING

Consider the execution of SIMPLESAMPLING on P. As mentioned earlier, for $\epsilon = 0.49$, any algorithm must return an optimal answer to fulfill $(1-\epsilon)$-approximation. However, if *neither* center point appears in the sample set S (obtained by SIMPLESAMPLING), an optimal answer on S will always cover a border point, in which case SIMPLESAMPLING fails to find an optimal answer on P (which should be at the center of the circle, as shown in Figure 18).

By the random nature of S, the probability for S to miss both center points is $\frac{(n-s)(n-s-1)}{n(n-1)}$. Therefore, SIMPLESAMPLING fails with probability at least $1-\delta$ when

$$\frac{(n-s)(n-s-1)}{n(n-1)} \geq 1-\delta$$

which holds as long as $s \leq (1-\sqrt{1-\delta})n - 1$. Therefore, s will have to be at least $(1-\sqrt{1-\delta})n = \Omega(n)$ so that SIMPLESAMPLING can succeed with probability at least δ.

Weighted Sampling with Replacement

Let G be a *ground set* of n elements, such that each element $e \in G$ carries a positive weight $w(e)$. Let $W = \sum_{e \in S} w(e)$. A *weighted random sample* refers to an element e' such that $\mathbf{Pr}[e' = e] = w(e)/W$ for every $e \in G$. The goal of *weighted sampling with replacement* is to obtain a multi-set S of s independent weighted random samples.

Next we describe an algorithm for obtaining S in $O(n + s \log s)$ time. First, generate independently a set X of s random numbers between 0 and 1. Then, sort these numbers in ascending order using $O(s \log s)$ time. Let the sorted order be $x_1, ..., x_s$.

Let $e_1, ..., e_n$ be the elements in G (in an arbitrary order). Scan G once to obtain W. Next, we generate a set I of intervals with another scan of G as follows. In the outset, $I = \emptyset$, and initialize an interval $[y, z]$ with $y = z = 0$. Then, we process each element of G in turn. Specifically, for e_i ($1 \leq i \leq n$), update $[y, z]$ by setting y to z, and increasing z by $w(e_i)/W$; let $I(e_i)$ be the resulting $[y, z]$, and add $I(e_i)$ to I. Note that all the intervals of $e_1, ..., e_n$ are disjoint, and their union equals $[0, 1]$. Furthermore, $I(e_1), ..., I(e_n)$ are sorted in ascending order of left end points. The computation of I takes $O(n)$ time.

Finally, e_i is added to the sample set S as many times as the number of values in X covered by $I(e_i)$ (if $I(e_i)$ contains no number in S, $e_i \notin S$). We can obtain, for all $I(e_i)$, the number $|X \cap I(e_i)|$ in $O(n + s)$ time, by "merging" sets I and X, utilizing the sorted orders of both sets. This completes the description of the algorithm.