Theoretical Computer Science 743 (2018) 130–147

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

# The range 1 query (R1Q) problem

Michael A. Bender [a,b], Rezaul A. Chowdhury [a], Pramod Ganapathi [a,*], Samuel McCauley [a], Yuan Tang [c]

[a] Department of Computer Science, Stony Brook University, NY, USA
[b] Tokutek, Inc., USA
[c] Software School, and, Shanghai Key Laboratory of Intelligent Information Processing, Fudan University, China

ARTICLE INFO

*Article history:*
Received 31 January 2015
Received in revised form 27 August 2015
Accepted 30 December 2015
Available online 7 January 2016

*Keywords:*
Range 1 query
R1Q
Range query
Range emptiness
Emptiness query
Existential query
Bit matrix
Randomized
Rectangular
Orthogonal
Non-orthogonal
Triangular
Polygonal
Simplicial
Circular
Spherical

ABSTRACT

We define the ***range 1 query (R1Q) problem*** as follows. Given a $d$-dimensional ($d \geq 1$) input bit matrix $A$ (consisting of 0's and 1's), preprocess $A$ so that for any given region $\mathcal{R}$ of $A$, efficiently answer queries asking if $\mathcal{R}$ contains a 1 or not. We consider both orthogonal and non-orthogonal shapes for $\mathcal{R}$ including rectangles, axis-parallel right-triangles, certain types of polygons, axis-parallel right simplices and spheres. We provide space-efficient deterministic and randomized algorithms with constant query times (in constant dimensions) for solving the problem in the word-RAM model. The space usage in bits is sublinear, linear, or near linear in the size of $A$, depending on the algorithm.



## 1. Introduction

Range searching is one of the most fundamental problems in computational geometry. Many geometric problems can be modeled as range searching problems. It arises in application areas including geographical information systems, computer graphics, computer aided design, spatial databases, and time series databases. Range searching encompasses different types of problems, such as range counting, range reporting, emptiness queries, and optimization queries. This problem has been extensively studied in literature [1–5].

The ***range 1 query (R1Q) problem*** is defined as follows. Given a $d$-dimensional ($d \geq 1$) input bit matrix $A$ (consisting of 0's and 1's), preprocess $A$ to efficiently answer queries asking if any given range $\mathcal{R}$ of $A$ is empty (does not contain a 1)

* Corresponding author.
*E-mail addresses:* bender@cs.stonybrook.edu (M.A. Bender), rezaul@cs.stonybrook.edu (R.A. Chowdhury), pganapathi@cs.stonybrook.edu (P. Ganapathi), smccauley@cs.stonybrook.edu (S. McCauley), yuantang@fudan.edu.cn (Y. Tang).

http://dx.doi.org/10.1016/j.tcs.2015.12.040
0304-3975/

or not, denoted by $\text{R1Q}_A(\mathcal{R})$ or simply $\text{R1Q}(\mathcal{R})$. For example, in 2-D, the range $\mathcal{R}$ can be a rectangle, a right triangle, a polygon, a circle, or any other shape.

In this paper, we investigate solutions to the R1Q problem in the ***word-RAM model***. If $N$ is the input size in bits, the machine word is assumed to be of size $\Omega(\log N)$ bits. We assume that accesses to memory cells, arithmetic operations $(+, -, \times, \div, \text{mod})$, comparisons $(<, >, =, \leq, \geq)$, bitwise operations (bitwise-AND, OR, XOR), and shifting a word left or right a specific number of bits each take $\Theta(1)$ time. Almost all our solutions can be easily implemented to use only bit shifts, bitwise ORs, comparisons, and memory accesses in the query execution algorithms. Our solutions are amenable to hardware that do not use complicated arithmetic operations i.e., additions, subtractions, multiplications, divisions, and mod.

The solutions we investigate share the following characteristics, if possible. First of all, we want queries to run in constant time (assuming $d$ is fixed), even for $d \geq 2$ dimensions. Second, we are interested in solutions that have space linear or sublinear in the number of bits in the input grid. Depending on the algorithm, we give solutions in one of the two models [6]: (*a*) indexing model (or systematic scheme), in which we retain the input matrix to be accessed by the query execution algorithms, and space generally refers to the additional space for the index data structure; or (*b*) encoding model (or non-systematic scheme), in which we discard the input matrix and access only the preprocessed data; Here, space refers to the total space. Note that while our sublinear bounds are parameterized by the number of 1's in the grid, this is still larger than the information-theoretic lower bounds. For our motivating applications, information-theoretically optimal space is less important than constant query time. Third, we are interested in grid inputs [7,8], where the input is a matrix consisting of 0's and 1's. Hence we view the problem in terms of pixels/voxels rather than as a set of spatial points. We take advantage of this grid perspective in our algorithms using table lookups and hashing to achieve constant-time queries. Finally, we are interested in both orthogonal and non-orthogonal queries, and we require solutions that are simple enough to be implementable.

### 1.1. Previous results

The R1Q problem can be solved using data structures such as balanced binary search trees, kd-trees, quad trees, range trees, partition trees, and cutting trees (see [1]), which take the positions of the 1-bits as input. It can also be solved using a data structure proposed by Overmars [7], which uses priority search trees, y-fast tries, and q-fast tries and takes the entire grid as input. However, in $d$-D ($d \geq 2$), in the worst case, these data structures have a query time at least polylogarithmic in the number of 1's in the grid and occupy a near-linear number of bits w.r.t. the number of 1's.

The R1Q problem can also be solved via the solutions of the range partial sum [9,10] and the range minimum query (RMQ) [11–21] problems. Though several efficient algorithms have been developed to solve the problem in 1-D and 2-D, their generalizations to 3-D and higher dimensions (while occupying a linear number of bits) are not known yet. Also, there is little work on space-efficient constant-time RMQ solutions for non-orthogonal ranges. Also, some of the above data structures that use trees and partial sum use semigroup arithmetic model, which is more restrictive than our model and hence the results cannot be compared fairly.

The R1Q problem can also be solved using rank queries. The data structures to answer rank queries in [22–31] either do not generalize to 2-D and higher dimensions or they take super-constant time for answering queries. Farzan et al. [32, 33] present space-efficient data structures to answer orthogonal rank queries in constant time for all dimensions. They present simple linear-space structures as well as more complicated structures occupying entropy-bounded space. Their data structures are more space-efficient than our orthogonal deterministic structures and answer the more general problem of rank queries.

There are two major differences between our solution and Farzan et al.'s method to solve the R1Q problem. First and foremost, the powers-of-2 approach that we use to answer orthogonal R1Q deterministically can be easily extended to answer orthogonal R1Q probabilistically using randomized algorithms in the encoding model. If we allow some errors, we can reduce the space to significantly sublinear size when the number of 1-bits is asymptotically smaller than the total number of bits, while answering the queries in constant time. The powers-of-2 approach can also be used to answer right-triangular R1Q (which forms the basis for some polygonal queries) in constant time, occupying near-linear space in bits, which is significantly less for right-triangular queries compared to the space consumption of standard approaches. On the other hand, it is not clear how Farzan et al.'s method can be extended to answer orthogonal R1Q probabilistically using sublinear space or to answer right-triangular R1Q. Second, almost all our query execution algorithms can be made to use the following four simple operations only: comparisons, bit shifts, bitwise ORs, and memory accesses (see Section 2) without asymptotic increase in space and time bounds. Hence they can be implemented on a simpler hardware without complicated arithmetic operations. However, Farzan et al.'s method uses additions, subtractions, and more expensive operations such as multiplications on the values. It is not straightforward to replace all those arithmetic operations that work on values with simpler non-arithmetic operations while retaining the linear-space and constant-time bounds.

### 1.2. Motivation

We encountered the R1Q and R0Q (whether a range contains a 0) problems while trying to optimize stencil computations [34] in the ***Pochoir*** stencil compiler [35,36], where we had to answer octagonal R1Q and octagonal R0Q on a static 2-D

```
UpdateRange(R)
1.  if !R0Q(R) do
2.      {all points in R satisfy P.}
3.      funcptr ← PUpdatePoint
4.  else if !R1Q(R) do
5.      {no points in R satisfy P.}
6.      funcptr ← NUpdatePoint
7.  else
8.      {not all points in R satisfy P.}
9.      funcptr ← UpdatePoint
10. for each grid point p in R do
11.     funcptr(p)
```

```
UpdatePoint(p)
1. {update p only if it satisfies P.}
2. if p.property = 1 do
3.     p.value ← new value
4. do some stuff
```

```
PUpdatePoint(p)
1. {p satisfies P. update p.}
2. p.value ← new value
3. do some stuff
```

```
NUpdatePoint(p)
1. {p doesn't satisfy P. don't update p.}
2. do some stuff
```

**Fig. 1.** Examples of the procedures in Pochoir that make use of R1Q and R0Q.

property grid. Stencil computations have applications in physics, computational biology, computational finance, mechanical engineering, adaptive statistical design, weather forecasting, clinical medicine, image processing, quantum dynamics, oceanic circulation modeling, electromagnetics, multigrid solvers, and many other areas (see the references in [35]).

In Fig. 1, we provide a simplified exposition of the problem encountered in Pochoir. There are two grids of the same size: a static property grid and a dynamic value grid. Each property grid cell is set to 1 if it satisfies some property $\mathcal{P}$ and 0 otherwise. When Pochoir needs to update a range $\mathcal{R}$ in the value grid (see UpdateRange), its runtime system checks whether all or none of the points in $\mathcal{R}$ satisfy $\mathcal{P}$ in the property grid, and based on the query result it uses an appropriate precompiled optimized version of the original code (see PUpdatePoint, NUpdatePoint) to update the range in the value grid. To check if all points in $\mathcal{R}$ satisfy $\mathcal{P}$, Pochoir uses R0Q($\mathcal{R}$), and to check if no points in $\mathcal{R}$ satisfy $\mathcal{P}$, it uses R1Q($\mathcal{R}$).

Pochoir needs time-, space-, and cache-efficient data structures to answer R1Q. The solutions should achieve constant query time and work in all dimensions. Although it is worth trading off space to achieve constant query times, space is still a scarce resource.

### *1.3. Our contributions*

We solve the R1Q problem for orthogonal and non-orthogonal ranges in the word-RAM model. Our major contributions as shown in Table 1 are as follows:

1. **[Orthogonal deterministic.]** We present two deterministic data structures to answer R1Q for orthogonal ranges in all dimensions and for any data distribution. They occupy linear space in bits and answer queries in constant time for any constant dimension.
2. **[Orthogonal randomized.]** We present randomized data structures to answer R1Q for orthogonal ranges. The structures occupy sublinear space in bits and provide a tradeoff between query time and error probability.
3. **[Non-orthogonal deterministic.]** We present deterministic data structures to answer R1Q for non-orthogonal shapes, such as axis-parallel right-triangles, certain simple polygons, axis-parallel right simplices, and spheres, for all dimensions. The structures occupy linear, near-linear, or subquadratic space in bits and answer queries efficiently.

We use techniques such as ***power hyperrectangles***, ***power right triangles***, ***sketches***, ***sampling***, ***the four Russians trick***, and ***compression*** in our data structures. A careful combination of these techniques allows us to solve a large class of R1Q problems. Techniques such as power hyperrectangles, table lookup, and the four Russians trick are already common in RMQ-style operations, while sketches, power right-triangles, and compression are not. Our algorithms make use of only bit shifts, bitwise ORs, comparisons, and memory accesses and hence are amenable to be implemented on a simpler hardware which does not support arithmetic operations.

### *1.4. Organization of the paper*

The paper is organized as follows. Section 2 presents deterministic and randomized algorithms to answer orthogonal R1Qs on a grid in constant time for constant dimensions. Section 3 presents deterministic algorithms to answer non-orthogonal R1Qs on a grid, for axis-parallel right triangles, some polygons, axis-parallel right simplices, and spheres.

## 2. Orthogonal range 1 queries

In this section, we present deterministic and randomized algorithms/data structures for answering orthogonal R1Qs in constant time and which occupy linear space in bits. Table 2 lists a few of the existing algorithms and/or data structures to solve the orthogonal R1Q problem.

**Table 1**
R1Q algorithms in this paper. Here, $N = n^d$ = total #bits, $N_1$ = #nonzero bits, and $N_0$ = #zero bits in the input hypercubic bit matrix $A$, and $d$ = #dimensions. In the indexing model, space refers to the additional space apart from the read-only input matrix.

| Shape | Space (in bits) | Time | Comments |
|---|---|---|---|
| Orthogonal (Deterministic) | | | |
| $d$-D | $\mathcal{O}\left((d+1)!\left(\frac{2}{\ln 2}\right)^d N\right)$ | $\mathcal{O}(4^d d)$ | for $d$ dimensions; encoding model |
| $d$-D | $\mathcal{O}\left(d!\left(\frac{2}{\ln 2}\right)^d N\right)$ | $\mathcal{O}(4^d d!)$ | for $d$ dimensions; indexing model |
| Orthogonal (Randomized) | | | |
| 1-D (sketch) | $\mathcal{O}\left(\sqrt{NN_1}\log N\log\frac{1}{\delta}\right)$ | $\mathcal{O}\left(\ln\frac{1}{\delta}\right)$ | $\delta\in\left(0,\frac{1}{4}\right)$; correct for range size $\geq\sqrt{N/N_1}$, otherwise correct with prob $\geq 1-4\delta$; extendible to $\geq$ 2-D; encoding model |
| 1-D (sketch) | $\mathcal{O}\left(\begin{pmatrix}N_1\log^3 N\log_{1+\gamma}\frac{1}{\delta}\\ +N^{\frac{1}{c}}\log\log N\end{pmatrix}\right)$ | $\mathcal{O}(\log\frac{c}{\delta})$ | $\gamma,\delta\in\left(0,\frac{1}{4}\right)$, integer $c>1$; with prob $\geq 1-4\delta$ at most $4\gamma$ fraction of all query results will be wrong; extendible to $\geq$ 2-D; encoding model |
| 1-D (sampling) | $\mathcal{O}(s)$ | $\mathcal{O}\left(\frac{1}{\varepsilon}\ln\frac{1}{\delta}\right)$ | $\varepsilon,\delta\in(0,1)$, $s=\Omega(\log N)$; always correct for range size $\geq(N\log N)/s$, otherwise correct with prob $\geq 1-\delta$ when $\geq\varepsilon$ fraction of all range entries are 1; extendible to $\geq$ 2-D; indexing model |
| Non-Orthogonal (Deterministic) | | | |
| Right triangles | $\mathcal{O}\left(N\log N+N_0\log^2 N\right)$ | $\mathcal{O}(1)$ | not extendible to $\geq$ 3-D; encoding model |
| Right simplices | $\mathcal{O}\left(N^{2-1/d}\log N\right)$ | $\mathcal{O}(d^2)$ | for $d$ dimensions; encoding model |
| Spheres | $\mathcal{O}(N)$ | $\mathcal{O}(d)$ | for $d$ dimensions; encoding model |

**Table 2**
A few algorithms and/or data structures that could be used to solve the orthogonal R1Q problem. Here, $N = n^d$ = total #bits and $N_1$ = #nonzero bits, $d$ = #dimensions, $\alpha(a,b)$ is the inverse Ackermann function, and $*$ represents amortized time. The term $H_k$ represents the $k$-th order entropy and $\sigma$ the alphabet size of the array. The last column represents the domain, which can be a grid ($\mathbb{Z}^d$) or real space ($\mathbb{R}^d$). Please refer to Table 1 for more details on the randomized variants.

| Data structure | Space (in bits) | Time | Dim |
|---|---|---|---|
| Trivial | $\mathcal{O}(N\log N)$ | $\mathcal{O}(2^d)$ | $\geq$ 1-D $\mathbb{Z}^d$ |
| Range trees [37,38] | $\mathcal{O}\left(N_1\log^d N_1\right)$ | $\mathcal{O}\left(\log^{d-1} N_1\right)$ | $\geq$ 2-D $\mathbb{R}^d$ |
| Chazelle's range trees [28] | $\mathcal{O}\left(N_1\frac{\log^d N_1}{\log^{d-1}\log N_1}\right)$ | $\mathcal{O}\left(\log^{d-1} N_1\right)$ | $\geq$ 2-D $\mathbb{R}^d$ |
| Overmars structure [7] | $\mathcal{O}\left(N_1\log^{d-1} N_1\sqrt{\log N}\right)$ | $\mathcal{O}\left(\log^{d-1} N_1\right)$ | $\geq$ 3-D $\mathbb{Z}^d$ |
| Yuan Atallah RMQ [11] | $\mathcal{O}\left(2^d d!N\log N\right)$ | $\mathcal{O}(3^d)$ | $\geq$ 2-D $\mathbb{Z}^d$ |
| Range partial sum [9] | $\mathcal{O}(N\log N)$ | $\mathcal{O}\left(\alpha(kN,N)^d\right)*$ | $\geq$ 2-D $\mathbb{Z}^d$ |
| Rank queries [22] | $\mathcal{O}\left(N\frac{\log\log N}{\log N}\right)$ | $\mathcal{O}(1)$ | 1-D $\mathbb{Z}^d$ |
| Dominance queries [39] | $\mathcal{O}\left(N_1\frac{\log^{d-1} N_1}{\log^{d-2}\log N_1}\right)$ | $\mathcal{O}\left(\left(\frac{\log N_1}{\log\log N_1}\right)^{d-1}\right)$ | $\geq$ 3-D $\mathbb{R}^d$ |
| Bit vectors [31] | $\mathcal{O}(nH_k+o(N\log\sigma))$ | $o\left((\log\log\sigma)^3\right)$ | 1-D $\mathbb{Z}^d$ |
| Rank queries [33] | $N+o(N)$ | $\mathcal{O}(4^d d)$ | $\geq$ 2-D $\mathbb{Z}^d$ |
| Rank queries [33] | $H+o(N)$ | $\mathcal{O}(4^d d)$ | $\geq$ 2-D $\mathbb{Z}^d$ |
| Our algorithm 1 | $\mathcal{O}\left((d+1)!\left(\frac{2}{\ln 2}\right)^d N\right)$ | $\mathcal{O}(4^d d)$ | $\geq$ 2-D $\mathbb{Z}^d$ |
| Randomized variant 1 | $\mathcal{O}\left(\sqrt{NN_1}\log N\log\frac{1}{\delta}\right)$ | $\mathcal{O}\left(\ln\frac{1}{\delta}\right)$ | 1-D $\mathbb{Z}^d$ |
| Randomized variant 2 | $\mathcal{O}\left(\begin{pmatrix}N_1\log^3 N\log_{1+\gamma}\frac{1}{\delta}\\ +N^{\frac{1}{c}}\log\log N\end{pmatrix}\right)$ | $\mathcal{O}(\log\frac{c}{\delta})$ | 1-D $\mathbb{Z}^d$ |
| Our algorithm 2 | $\mathcal{O}\left(d!\left(\frac{2}{\ln 2}\right)^d N\right)$ | $\mathcal{O}(4^d d!)$ | $\geq$ 2-D $\mathbb{Z}^d$ |

Almost all our solutions use only the following four simple operations: bit shifts, bitwise ORs, comparisons, and memory accesses. For simplicity of exposition, we occasionally use the arithmetic operations to explain our algorithms. But, the arithmetic operations in all our query algorithms can be replaced with constant number of bit shifts, bitwise ORs, comparisons, and memory accesses without asymptotic increase in the space and time bounds using a very simple data structure constructed as follows. In $d$-dimensions, where $d\geq 3$, the arithmetic operation $i\oplus j$, where $\oplus\in\{+,-,\times,\div,\text{mod}\}$ and the values $i$, $j$, and $i\oplus j$ are from the domain and range $[1,\mathcal{O}\left(N^{1/d}\right)]$, can be answered in $\mathcal{O}(1)$ by storing all possible values

of $i$, $j$, and $i \oplus j$. Note that the maximum value of an input or a result of an arithmetic operation is $\mathcal{O}\left(N^{1/d}\right)$. The space required to store all input and results is $\mathcal{O}\left((1/d)N^{2/d}\log N\right)$ bits and for $d \geq 3$ it is sublinear with respect to the size of the input matrix to the R1Q algorithm.

The algorithms in this paper rely on finding the most significant bit (MSB) of positive integers less than $\mathcal{O}\left(N^{1/d}\right)$ in constant time without using arithmetic operations. Several classical solutions can be used to compute the MSB of a positive integer using space independent of $N$ in constant time [40–44]. However, the previous approaches are quite complicated and make use of several arithmetic operations. As we would like to avoid arithmetic operations, here we give a lemma that is relatively simple and uses only three simple operations in the query execution: bit shifts, comparisons, and memory accesses using space sublinear in the size of the input matrix to the R1Q algorithm for $d \geq 3$.

**Lemma 1.** *Given integers $M \in [1, 2^w)$ and $r$, where $r$ is a constant and a power of 2 in $[2, w)$, in the word-RAM model with $w$-bit words, one can construct a table occupying $\mathcal{O}\left(M^{1/r}\log\log M\right)$ bits of space to answer MSB queries for integers in $[1, M]$ in $\mathcal{O}(r)$ time requiring only bit shifts, comparisons, and memory accesses.*

**Proof.** We need to find the MSB of a $\lceil \log M \rceil$-bit integer $m \in [1, M]$. The procedure is as follows.

*Preprocessing.* Compute and store the value of $\lceil \log M \rceil$ using brute force. Divide the $\lceil \log M \rceil$ bits into $r \in [2, w)$ blocks, each of size $\lceil \lceil \log M \rceil / r \rceil$ bits. Division (resp. multiplication) by a power of 2 can be implemented using bit shifts. At most $2^{\lceil \lceil \log M \rceil / r \rceil} = \mathcal{O}\left(M^{1/r}\right)$ different numbers are possible using $\lceil \lceil \log M \rceil / r \rceil$ bits. Store the MSB location for each of these $\mathcal{O}\left(M^{1/r}\right)$ numbers in a table. The total space requirement is $\mathcal{O}\left(M^{1/r}\log\log M\right)$ bits. We make use of the constant $r$ during query execution.

*Query execution.* There are two stages in finding the MSB of $m$ represented using $\lceil \log M \rceil$ bits: ($a$) Find the block: Using bit shifts, we can find the exact block of size $\lceil \lceil \log M \rceil / r \rceil$ bits in which the MSB is present in $\mathcal{O}(r)$ time. ($b$) Find the bit: Once the block that has the MSB is found, we simply use the bits in that block as an index to the already stored table that gives us the actual MSB. □

### 2.1. Preliminaries: deterministic 1-D algorithm

The input is a bit vector $A[0 \ldots N-1]$, where $N \in [1, 2^w)$ and $w$ is the word size. The query $\text{R1Q}_A(i, j)$, where $i \leq j$, asks if there exists a 1 in the subarray $A[i \ldots j]$. For simplicity, assume $N$ to be an even power of 2.

*Preprocessing*

Array $A$ has $M = \lceil \frac{N}{w} \rceil$ words. For each $p \in [0, \log M]$, we construct arrays $L_p$ and $R_p$ of size $\frac{M}{2^p}$ words each. Let $W(i)$ denote the $i$th ($i \in [0, M-1]$) word in $A$. Then, $L_p$ is defined as follows: $L_0[i]$ is 0, if $W(i)$ has a 1, and 1 if $W(i)$ does not contain a 1.

$$L_{p(\geq 1)}[i] = \begin{cases} L_{p-1}[2i] & \text{if } L_{p-1}[2i] < 2^{p-1}, \\ 2^{p-1} + L_{p-1}[2i+1] & \text{otherwise.} \end{cases}$$

The $R_p$ array can be computed similarly. The array element $L_p[i]$ (and $R_p[i]$) stores the distance of the leftmost (respectively, rightmost) word that contains a 1 in the $i$th block of $2^p$ contiguous words of $A$, measured from the start (and end) of the block. The value $L_p[i] = 2^p$ ($R_p[i] = 2^p$) means that the $i$th block of $2^p$ contiguous words of $A$ does not contain a 1.

*Query execution*

To answer $\text{R1Q}_A(i, j)$, we consider two cases: (1) *Intra-word queries:* If $(i, j)$ lies inside one word, we answer R1Q using bit shifts. (2) *Inter-word queries:* If $(i, j)$ spans multiple words, then the query gets split into three subqueries: ($a$) R1Q from $i$ to the end of its word, ($b$) R1Q of the words between $i$'s and $j$'s word (both exclusive), and ($c$) R1Q from the start of $j$'s word to $j$.

The answer to an inter-word query is 1 if and only if the R1Q for at least one of the three subqueries is 1. The first and third subqueries are intra-word queries and can be answered using bit shifts. Let the indices of the words containing indices $i$ and $j$ be $I$ and $J$, respectively. Then, the second subquery, denoted by $\text{R1Q}_{L_0}(I+1, J-1)$, is answered as follows. Using the MSB of $J - I - 1$, we find the largest integer $p$ such that $2^p \leq J - I - 1$. The query $\text{R1Q}_{L_0}(I+1, J-1)$ is then decomposed into the following two overlapping queries of size $2^p$ each: $\text{R1Q}_{L_0}(I+1, I+2^p)$ and $\text{R1Q}_{L_0}(J-2^p, J-1)$. If either of those two ranges contains a 1 then the answer to the original query will be 1, and 0 otherwise. We show below how to answer $\text{R1Q}_{L_0}(I+1, I+2^p)$. Query $\text{R1Q}_{L_0}(J-2^p, J-1)$ is answered similarly.

Split $L_0$ into blocks of size $2^p$. Then, the range $\text{R1Q}_{L_0}(I+1, I+2^p)$ can be covered by one or two consecutive blocks. Let's say that the word with index $I+1$ is in the $k$th block. If the range lies in one block, we find whether a 1 exists in that block by checking whether $L_p[k] < 2^p$ is true. If the range is split across two consecutive blocks, we find whether a 1 exists in at least one of the two blocks by checking whether at least one of $R_p[k] \leq (k+1)2^p - I$ and $L_p[k+1] \leq I + 2^p - (k+1)2^p$ is true.

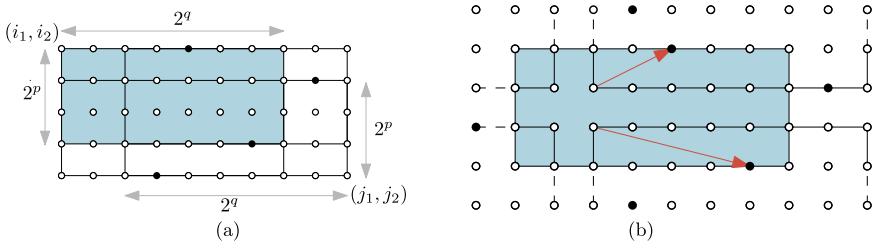


**Fig. 2.** Rectangles: (*a*) Query rectangle split into four possibly overlapping power rectangles. (*b*) Power rectangle divided into four regions by four split rectangles.

*Complexity analysis*

We analyze the time, space, and query complexities below.

*Extra space.* Space is given in bits unless stated otherwise. We store $L_p$ and $R_p$ arrays. Let $M = \lceil \frac{N}{w} \rceil$. For each $p \in [0, \log M]$, the $L_p$ (resp. $R_p$) array stores $\frac{M}{2^p}$ entries occupying $p+1$ bits each. Hence, the total space occupied by all $L_p$ (resp. $R_p$) arrays is $\Theta\left(\sum_{p=0}^{\log M} \frac{M}{2^p}(p+1)\right) = \Theta(M) = \mathcal{O}(N/\log N)$. We use $\mathcal{O}\left(\sqrt{N}\log\log N\right)$ bits for the MSB table to support MSB queries in constant time (see Lemma 1). Thus the space complexity is $\mathcal{O}(N/\log N)$ bits.

*Preprocessing time.* Preprocessing the $L_p$ and $R_p$ tables for $p = 0$ takes $\mathcal{O}(N/\log N)$ time as we scan $A$ once. For each of $p \in [1, \log M]$, constructing the two tables using dynamic programming takes $\Theta\left(\frac{M}{2^p}\right)$ time. Hence, the total time to construct the two tables is $\Theta\left(M + \sum_{p=1}^{\log M} \frac{M}{2^p}\right) = \Theta(M) = \mathcal{O}(N/\log N)$ which dominates the cost of constructing the MSB table.

*Query time.* The query time is dominated by finding the largest power of two less than or equal to a particular integer, which takes $\Theta(1)$ time using the MSB table of size $\mathcal{O}\left(\sqrt{N}\log\log N\right)$. The remainder of the query is performed using $\Theta(1)$ comparisons and table lookups.

### 2.2. Deterministic d-D algorithm

For $d$-D ($d \geq 2$) R1Q, the input is a hypercubic bit matrix $A$ of size $N = n^d$. Here we present an algorithm for a 2-D matrix of size $N = n \times n$, but the algorithm extends to higher dimensions. For simplicity, we assume $n$ to be a power of 2. The query R1Q($[i_1, j_1][i_2, j_2]$) asks if there exists a 1 in the submatrix $A[i_1 \ldots j_1][i_2 \ldots j_2]$.

*Preprocessing*

For each $p, q \in [0, \log n]$, we partition $A$ into $\frac{n}{2^p} \times \frac{n}{2^q}$ blocks, each of size $2^p \times 2^q$. We call these $(p, q)$-blocks. For each $(p, q)$ pair, we construct four tables of size $\frac{N}{2^{p+q}} \times \min(2^p, 2^q)$ each:

(i) $TL_{p,q}$: if $p \leq q$, $TL_{p,q}[i, j][k]$ indicates that any rectangle of height $k \in [0, 2^p)$ starting from the top-left corner of the current block must have width at least $TL_{p,q}[i, j][k]$ in order to include at least one 1-bit.
(ii) $BL, TR, BR$: similar to $TL$ but starts from the bottom-left, top-right and bottom-right corners, respectively.

In all cases, a stored value of $\max(2^p, 2^q)$ indicates that the block has no 1.

*Query execution*

Given a query $[i_1, j_1][i_2, j_2]$, we find the largest integers $p$ and $q$ such that $2^p \leq j_1 - i_1 + 1$ and $2^q \leq j_2 - i_2 + 1$. The original query range can then be decomposed into four overlapping $(p, q)$-blocks, which we call ***power rectangles***, each with a corner at one of the four corners of the original rectangle, as shown in Fig. 2(*a*). If any of these four rectangles contains a 1, the answer to the original query will be 1, and 0 otherwise. We show below how to answer an R1Q for a power rectangle.

We consider the partition of $A$ into preprocessed $(p, q)$-blocks. It is easy to see that each of the four power rectangles of size $2^p \times 2^q$ will intersect at most four preprocessed $(p, q)$-blocks. We call each rectangle contained in both the power rectangle and a $(p, q)$-block a ***split rectangle*** (see Fig. 2(*b*)). The R1Q for a split rectangle can be answered using a table lookup, checking if the table values of the appropriate $(p, q)$-blocks are inside the power rectangle boundary, as shown in Fig. 2(*b*).

*Complexity analysis*

We analyze the time, space, and query complexities below.

*Space complexity.* In 2-D, for $p, q \in [0, \log n]$, the number of blocks of size $2^p \cdot 2^q$ is $\frac{N}{2^{p+q}}$. For each block, there are four tables that stores $2^{\min(p,q)}$ entries each, and each entry occupies $\max(p, q) + 1$ bits of space. The space occupied by the four

tables for all blocks of size $2^p \cdot 2^q$ is $S(p,q) = \left(\frac{4N}{2^{p+q}}\right) 2^{\min(p,q)} (\max(p,q)+1)$. Hence, the total space is $\sum_{p,q} S(p,q) = \Theta(N)$ bits. In $d$-D, the total space required for the algorithm for a given input matrix of size $N = n^d$ with $n \in \mathbb{Z}^+$ is computed as follows. The space occupied by $2^d$ tables for all blocks of size $2^{p_1} \cdot 2^{p_2} \cdots 2^{p_d}$, where $p_i \in [0, \log n]$ is

$$S(p_1, p_2, \ldots, p_d) = 2^d \cdot \frac{n}{2^{p_1}} \cdot \frac{n}{2^{p_2}} \cdots \frac{n}{2^{p_d}} \cdot \frac{2^{p_1+p_2+\cdots+p_d}}{2^{\max(p_1,p_2,\ldots,p_d)}} \cdot (\max(p_1, p_2, \ldots, p_d)+1)$$

as the number of bits of information stored per block, per table, for different heights of different dimensions in a block is the logarithm of the largest dimension of a block plus one i.e., $\max(p_1, p_2, \ldots, p_d)+1$. So,

$$\text{Total space} = \sum_{p_1=0}^{\log n} \sum_{p_2=0}^{\log n} \cdots \sum_{p_d=0}^{\log n} 2^d n^d \cdot \frac{\max(p_1, p_2, \ldots, p_d)}{2^{\max(p_1,p_2,\ldots,p_d)}}$$

We know that $\max(p_1, p_2, \ldots, p_d)$ can be any value from 0 to $\log n$. With this observation, the expression above with $d$ summations can be written as an expression with a single summation by grouping all $d$-tuples having a maximum value of $i$, for $i \in [0, \log n]$. Thus,

$$\begin{aligned}
\text{Total space} &= 2^d N \cdot \sum_{i=0}^{\log n} \frac{i}{2^i} \cdot \text{frequency of } i \text{ being the max in } d\text{-tuple} \\
&= 2^d N \cdot \sum_{i=0}^{\log n} \frac{i}{2^i} \cdot \left(d(i+1)^{d-1}\right) < 2^d \cdot dN \cdot \sum_{i=0}^{\log n} \frac{(i+1)^d}{2^i} \\
&= 2^{d+1} \cdot dN \cdot \sum_{i=1}^{\log n+1} \frac{i^d}{2^i} < 2^{d+1} \cdot dN \cdot \sum_{i=1}^{\infty} \frac{i^d}{2^i} \qquad \text{(OEIS A000629 [45])} \\
&= 2^{d+1} \cdot dN \cdot \mathcal{O}\left(\frac{d!}{\ln^{d+1} 2}\right) = \mathcal{O}\left((d+1)!\left(\frac{2}{\ln 2}\right)^d N\right) \text{ bits.}
\end{aligned}$$

*Preprocessing time.* In 2-D, for each $p, q \in [0, \log n]$, constructing the four tables using dynamic programming takes $\Theta\left(\frac{N}{2^{p+q}} \cdot 2^{\min(p,q)}\right) = \Theta(N)$ time. In $d$-D, the total preprocessing time to construct the data structure for a hypercubic matrix of size $N = n \times n \times \cdots \times n$ is given below.

$$\begin{aligned}
\text{Prep. time} &= \sum_{p_1=0}^{\log n} \sum_{p_2=0}^{\log n} \cdots \sum_{p_d=0}^{\log n} 2^d \cdot \frac{n^d}{2^{\max(p_1,p_2,\ldots,p_d)}} \\
&= 2^d N \cdot \sum_{i=0}^{\log n} \frac{1}{2^i} \cdot \text{frequency of } i \text{ being the max in } d\text{-tuple} \\
&= 2^d N \cdot \sum_{i=0}^{\log n} \frac{1}{2^i} \cdot \left(d(i+1)^{d-1}\right) < 2^{d+1} \cdot dN \cdot \sum_{i=1}^{\infty} \frac{i^{d-1}}{2^i} \quad \text{(OEIS A000629)} \\
&= \mathcal{O}\left(d!\left(\frac{2}{\ln 2}\right)^d N\right).
\end{aligned}$$

We use dynamic programming to compute the information for all tables for all blocks. The expression above can be simplified using the analysis of the total space. After simplification, the total preprocessing time is $\mathcal{O}\left(d!\left(\frac{2}{\ln 2}\right)^d N\right)$.

*Query time.* In 2-D, we access the tables at most $4^2$ times. Hence, the query time is $\mathcal{O}(1)$. For $d$-D, there are at most $2^d$ power hyperrectangles, each power hyperrectangle can have at most $2^d$ split hyperrectangles, and accessing each coordinate takes $\Theta(d)$ time. Hence, the query time is $\mathcal{O}(4^d d)$.

Overall, we have the following theorem.

**Theorem 1.** *Given a $d$-D input hypercubic grid of size $N = n^d$, each orthogonal R1Q on the grid can be answered deterministically in $\mathcal{O}\left(4^d d\right)$ time after preprocessing the grid in $\mathcal{O}\left(d!\,(2/\ln 2)^d N\right)$ time using $\mathcal{O}\left((d+1)!\,(2/\ln 2)^d N\right)$ bits of space in the encoding model. In 1-D, in the indexing model, the space can be reduced to $\mathcal{O}(N/\log N)$ bits.*

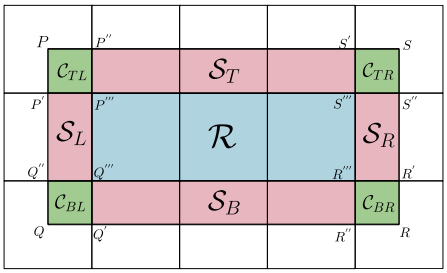


**Fig. 3.** A query rectangle split into constant number of query rectangles.

### 2.3. Recursive deterministic d-D algorithm

For $d$-D ($d \geq 2$) R1Q, the input is a bit matrix $A$ of size $N = n_1 \times n_2 \times \cdots \times n_d$. Here we give a recursive algorithm for a 2-D matrix of size $N = n \times n$, but the algorithm extends to higher dimensions. For simplicity, we assume $n$ to be a power of 2. The query R1Q($[i_1, j_1][i_2, j_2]$) asks if there exists a 1 in the submatrix $A[i_1 \ldots j_1][i_2 \ldots j_2]$.

*Preprocessing*

We create three separate structures:

(1) Let $\ell = \lceil \sqrt{(\log N)/2} \rceil$. We partition $A$ into $\lceil n/\ell \rceil \times \lceil n/\ell \rceil$ blocks, each of size $\ell \times \ell$; if a block is not of size $\ell \times \ell$ (e.g., some of the boundary blocks) then it is padded with 0's to make it an $\ell \times \ell$ block. A block with its top-left corner at $(i\ell, j\ell)$, where $i, j \in \{0, \ldots, \lceil n/\ell \rceil\}$, is called an $(i, j)$-block. Create a matrix $B$ such that $B[i][j]$ denotes if there exists a 1-bit in the $(i, j)$-block.

(2) Let $I = i\ell$, $I' = I + \ell - 1$, $J = j\ell$, and $J' = J + \ell - 1$. We create four $\lceil n/\ell \rceil \times \lceil n/\ell \rceil \times \ell$ matrices $S_L, S_R, S_T$, and $S_B$ as follows.

$$S_L[i, j][k] = \text{R1Q}_A([I, J + k], [I', J']), \qquad S_R[i, j][k] = \text{R1Q}_A([I, J], [I', J + k])$$
$$S_T[i, j][k] = \text{R1Q}_A([I + k, J], [I', J']), \qquad S_B[i, j][k] = \text{R1Q}_A([I, J], [I + k, J'])$$

where $i, j \in [0, \lceil n/\ell \rceil - 1]$ and $k \in [0, \ell - 1]$.

(3) Every block of $A$ is denoted by a sequence of $\ell^2$ bits of information. The total number of unique blocks of $A$ is $2^{\ell^2} \leq N^{3/4}$. For each such unique block we store an $\ell \times \ell$ lookup table to answer R1Q queries in that particular block.

*Query execution*

We want to answer the query R1Q($[i_1, j_1][i_2, j_2]$). The range query is depicted in Fig. 3. The corners of the query rectangle are $P(i_1, i_2)$, $Q(j_1, i_2)$, $R(j_1, j_2)$, and $S(i_1, j_2)$. The bottom-right, top-right, top-left, and bottom-left corners of the blocks containing $P$, $Q$, $R$, and $S$ are denoted by $P'''$, $Q'''$, $R'''$, and $S'''$, respectively. The query rectangle $PQRS$ is divided into queries $\mathcal{R}$, $\mathcal{S}_L, \mathcal{S}_R, \mathcal{S}_T, \mathcal{S}_B$, $\mathcal{C}_{TL}, \mathcal{C}_{BL}, \mathcal{C}_{TR}, \mathcal{C}_{BR}$. The region $\mathcal{R}$ is the rectangle $P'''S'''R'''Q'''$, the region $\mathcal{S}_T$ is the rectangle $P''S'S'''P'''$, the region $\mathcal{C}_{TL}$ is the rectangle $PP''P'''P'$, and other regions can be defined similarly. The R1Q for query $\mathcal{R}$ can be answered using the matrix $B$ in constant time using the data structures presented in Section 2.2. Let

$$PX = \lfloor i_1/\ell \rfloor, \qquad QX = \lfloor j_1/\ell \rfloor, \qquad RX = \lfloor j_1/\ell \rfloor, \qquad SX = \lfloor i_1/\ell \rfloor$$
$$PY = \lfloor i_2/\ell \rfloor, \qquad QY = \lfloor i_2/\ell \rfloor, \qquad RY = \lfloor j_2/\ell \rfloor, \qquad SY = \lfloor j_2/\ell \rfloor$$
$$LK = i_2 \bmod \ell, \qquad RK = j_2 \bmod \ell, \qquad TK = i_1 \bmod \ell, \qquad BK = j_1 \bmod \ell$$

where $(PX, PY)$, $(QX, QY)$, $(RX, RY)$, and $(SX, SY)$ are the coordinates of the blocks that contain points $P$, $Q$, $R$, and $S$, respectively. Then, the R1Q for queries $\mathcal{S}_L, \mathcal{S}_R, \mathcal{S}_T, \mathcal{S}_B$ can be answered recursively in constant time using 1-D range queries R1Q($S_L([(PX + 1) \ldots (QX - 1), PY][LK])$), R1Q($S_R([(SX + 1) \ldots (RX - 1), RY][RK])$), R1Q($S_T([PX, (PY + 1) \ldots (SY - 1)][TK])$), and R1Q($S_B([QX, (QY + 1) \cdots (RY - 1)][BK])$), respectively. We use our algorithm described in Section 2.1 to solve the 1-D range queries.

The R1Q for queries $\mathcal{C}_{TL}, \mathcal{C}_{BL}, \mathcal{C}_{TR}, \mathcal{C}_{BR}$ can be answered by $\mathcal{O}(1)$ lookups in the lookup table for the blocks containing $P$, $Q$, $R$, and $S$. Each block in the matrix $A$ is of size $\ell$ and hence there are only $2^{\ell^2} \leq N^{3/4}$ unique blocks and each block contains a lookup table to answer orthogonal queries inside it.

*Complexity analysis*

The space and time complexities of the algorithm are given below.

*Space complexity.* Let $S(n,d)$ be the total space complexity for $d$-D matrix of size $N = n^d$. To answer the intra-block queries we make use of the four Russians trick, requiring linear (i.e. $\Theta(n^d)$) bits of space. The analysis of the space required for matrix $B$ is given in Section 2.2. The $d$-D query range depends on at most $2d$ query ranges in $(d-1)$ dimensions as there are $2d$ faces in a $d$-D hypercube and they are preprocessed from different directions. We need to maintain $n$ such $(d-1)$-dimensional data structures for different heights. Also, as we saw in the preprocessing stage, the $(d-1)$-dimensional structures are used for inter-block queries (each block is of size $\ell^d$, where $\ell = \lceil((\log N)/2)^{1/d}\rceil$) and they reduce the parameter $n$ in $d$-D to $\frac{n}{((\log N)/2)^{1/d}}$ in $(d-1)$-dimension by taking the bitwise OR of the bits in a single dimension i.e., storing a 1 if any bit along that dimension is 1. Therefore, the recurrence for $S(n,d)$ is

$$S(n,d) = \begin{cases} \mathcal{O}(n) & \text{if } d = 1. \\ \Theta(n^d) + \mathcal{O}\left(d!\left(\frac{2}{\ln 2}\right)^d \frac{n^d}{\frac{1}{2}\log n^d}\right) + 2dn \cdot S\left(\frac{n}{(\frac{1}{2}\log n^d)^{1/d}}, d-1\right) & \text{if } d > 1. \end{cases}$$

We use induction to prove that $S(n,d) = \mathcal{O}\left(d!(2/\ln 2)^d n^d\right)$. Base case: When $d = 1$, $S(n,1) = \mathcal{O}(n)$, which is true by definition. Induction: Assume that $S(n',d-1) = \mathcal{O}\left((d-1)!(2/\ln 2)^{d-1} n^{d-1}\right)$. We would like to show that $S(n,d) = \mathcal{O}\left(d!(2/\ln 2)^d n^d\right)$. Let $c_1$ and $c_2$ be constants that upper bound the asymptotic terms in the recurrence; thus the first term is upper bounded by $c_1 n^d$ and the second is upper bounded by $2c_2 d!(2n/\ln 2)^d/\log n^d$. Let $c_3$ parameterize the solution to our recurrence; we will prove that $S(n,d) \le c_3 d!(2/\ln 2)^d n^d$.

$$\begin{aligned}
S(n,d) &\le c_1 n^d + c_2 d!\left(\frac{2}{\ln 2}\right)^d \frac{n^d}{\frac{1}{2}\log n^d} + 2dn \cdot S\left(\frac{n}{(\frac{1}{2}\log n^d)^{1/d}}, d-1\right) \\
&\le c_1 n^d + 2c_2 \frac{d!}{\log n^d}\left(\frac{2}{\ln 2}\right)^d n^d + 2dnc_3\left((d-1)!\left(\frac{2}{\ln 2}\right)^{d-1}\left(\frac{n}{(\frac{1}{2}\log n^d)^{1/d}}\right)^{d-1}\right) \\
&= c_1 n^d + \frac{2c_2}{d\log n} d!\left(\frac{2}{\ln 2}\right)^d n^d + \frac{c_3 \ln 2}{\left(\frac{d}{2}\log n\right)^{\frac{d-1}{d}}} d!\left(\frac{2}{\ln 2}\right)^d n^d \\
&\le \left(\left(1 + \frac{2c_2}{d\log n}\right) + \left(\frac{c_3 \ln 2}{\left(\frac{d}{2}\log n\right)^{\frac{d-1}{d}}}\right)\right) d!\left(\frac{2}{\ln 2}\right)^d n^d \\
&\le ((1+c_2) + (c_3 \ln 2))\, d!\left(\frac{2}{\ln 2}\right)^d n^d \qquad (\text{for } n \ge 2, d \ge 2) \\
&\le (c_3(1-\ln 2) + c_3 \ln 2)\, d!\left(\frac{2}{\ln 2}\right)^d n^d \qquad \left(\text{for } c_3 \ge \frac{1+c_2}{1-\ln 2}\right) \\
&\le c_3 d!\left(\frac{2}{\ln 2}\right)^d n^d = \mathcal{O}\left(d!\left(\frac{2}{\ln 2}\right)^d n^d\right)
\end{aligned}$$

Thus, if we choose $c_3$ such that $c_3 \ge (1+c_2)/(1-\ln 2)$, we have $S(n,d) = \mathcal{O}\left(d!\left(\frac{2}{\ln 2}\right)^d n^d\right)$ and the recurrence is satisfied.

*Preprocessing time.* The matrices $B, S_L, S_R, S_T$, and $S_B$ can be built in $\Theta(N)$ time using dynamic programming. The lookup tables for the unique blocks can be built in $\mathcal{O}\left(N^{3/4}\left(\sqrt{(\log N)/2}\right)^2\right) = \mathcal{O}\left(N^{3/4}\log N\right)$ time.

*Query time.* The $d$-dimensional query range depends on at most $2d$ query ranges in $(d-1)$ dimensions, as the number of faces in a $d$-D hypercube is $2d$. Also, at every dimension, the time required to answer the query ranges for the blocks at the corners of a hypercube should be considered and there are $2^d$ corners for a hypercube. Answering the region $\mathcal{R}$ of a query range using the structures of Section 2.2 requires $\mathcal{O}(4^d d)$ time. Hence, the query time $T(d)$ for $d$-D can be computed using the recurrence

$$T(d) = \begin{cases} \Theta(1) & \text{if } d = 1, \\ \mathcal{O}(2^d d) + \mathcal{O}(4^d d) + 2d \cdot T(d-1) & \text{if } d > 1. \end{cases}$$

We use induction to prove that $T(d) = \mathcal{O}(4^d d!)$. Basis: When $d = 1$, $T(1) = \mathcal{O}(1)$, which is true by definition. Induction: Assume that $T(d-1) = \mathcal{O}(4^{d-1}(d-1)!)$ for some $d > 1$. We like to prove that $T(d) = \mathcal{O}(4^d d!)$. From the recurrence,

$$\begin{aligned} T(d) &\le c_1 4^d d + 2d \cdot T(d-1) \qquad \text{(for some constant } c_1\text{)} \\ &\le c_1 4^d d + 2dc_2 \cdot 4^{d-1}(d-1)! \qquad \text{(for some constant } c_2\text{)} \\ &= c_1 4^d d + (c_2/2) \cdot 4^d d! = \mathcal{O}\left(4^d d!\right). \end{aligned}$$

**Theorem 2.** *Given a d-D input hypercubic bit matrix of size $N = n \times n \times \cdots \times n$, each orthogonal R1Q on the grid can be answered deterministically in $\mathcal{O}\left(4^d d!\right)$ time after preprocessing the grid in linear time (in the size of the input matrix) in the indexing model using $\Theta\left(d!(2/\ln 2)^d N\right)$ bits of space.*

### 2.4. Randomized algorithms

In this section, we present randomized algorithms that build on the deterministic algorithms given in Sections 2.1 and 2.2. We describe the algorithms for one dimension only. Extensions to higher dimensions are straightforward.

#### 2.4.1. Sketch-based algorithms

Our algorithms provide probabilistic guarantees based on the Count-Min (CM) sketch data structure proposed in [46]. Let $N_1$ be the number of 1-bits in the input bit array $A[0 \ldots N-1]$ for any data distribution. Then, the preprocessing time and space complexities depend on $N_1$ while the query time remains constant.

A CM sketch with parameters $\varepsilon \in (0,1]$ and $\delta \in (0,1)$ can store a summary of any given vector $\vec{a} = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ with $a_i \ge 0$ in only $\lceil \frac{e}{\varepsilon} \rceil \lceil \ln \frac{1}{\delta} \rceil \log ||\vec{a}||_1$ bits of space (where $e = 2.7182\ldots$ is the Euler's number and $||\vec{a}||_1$ (or $||\vec{a}||$) $= \sum_{i=0}^{n-1} a_i$), and can provide an estimate $\hat{a}_i$ of any $a_i$ with the following guarantees: $a_i \le \hat{a}_i$, and with probability at least $1-\delta$, $\hat{a}_i \le a_i + \varepsilon ||\vec{a}||_1$. It uses $t = \lceil \ln \frac{1}{\delta} \rceil$ hash functions $h_1 \ldots h_t : \{0 \ldots n-1\} \to \{1 \ldots b\}$ chosen uniformly at random from a pairwise-independent family, where bucket size $b = \lceil \frac{e}{\varepsilon} \rceil$. These hash functions are used to update a 2-D matrix $c[1:t][1:b]$ of $bt$ counters initialized to 0. For each $i \in [0, n-1]$ and each $j \in [1, t]$ one then updates $c[j][h_j(i)]$ to $c[j][h_j(i)] + a_i$. After the updates, an estimate $\hat{a}_i$ for any given query point $a_i$ is obtained as $\min_{1 \le j \le t} c[j][h_j(i)]$.

*Preprocessing*

In the deterministic algorithms we first compressed the input array by converting each word into a single bit, and then constructed $L_0$ and $R_0$ arrays from the compressed array. In the current algorithm we build the $L_0$ and $R_0$ arrays directly from the uncompressed input. For $p \in \left[0, \frac{1}{2}\log(N/N_1)\right]$, the $L_p$ and $R_p$ arrays are stored as CM sketches while for $p \in \left[\frac{1}{2}\log(N/N_1) + 1, \log N\right]$, the arrays are stored directly as in the deterministic case. Each $L_p[i]$ is added as $\left(L_p[i] + 1\right) \bmod \left(2^p + 1\right)$ to the CM sketch (similarly for $R_p[i]$). Thus a nonzero entry (of value at most $2^p$) is added to the CM sketch provided the corresponding block contains a 1, otherwise nothing is added. As a result for any given $L_p$ summation of all entries added to the CM sketch is at most $N_1 \times 2^p$, and we set $\varepsilon = \frac{1}{2 \times N_1 \times 2^p}$ for that sketch.

*Query execution*

Given a query $\text{R1Q}_A(i, j)$, we use the MSB of $j - i + 1$ to find the largest value of $p$ with $2^p \le j - i + 1$, and then follow the approach for answering case $(b)$ of inter-word queries described in Section 2.1. If $2^p > \sqrt{N/N_1}$, we use $L_p$ and $R_p$ arrays to answer the query correctly, otherwise we use the $L_p$ and $R_p$ values obtained from the corresponding CM sketches.

*Complexity analysis*

We compute space usage, preprocessing time and query time below.

*Total space.* Recall that a CM sketch with parameters $\varepsilon$ and $\delta$ occupies $\mathcal{O}\left(\frac{1}{\varepsilon} \ln \frac{1}{\delta} \log ||\vec{a}||\right)$ bits of space. For $p \in \left[0, \frac{1}{2}\log(N/N_1)\right]$, we store the $L_p$ and $R_p$ arrays as CM sketches with $\varepsilon = \frac{1}{2 \times N_1 \times 2^p}$. Hence, the CM sketches occupy $\mathcal{O}\left(\sum_{p=0}^{\lfloor \frac{1}{2}\log(N/N_1) \rfloor} \left(2N_1 \cdot 2^p \ln\left(\frac{1}{\delta}\right) \log\left(N_1 \cdot 2^p\right)\right)\right) = \mathcal{O}\left(\sqrt{NN_1} \log N \log\left(\frac{1}{\delta}\right)\right)$ bits of space. For $p \in \left[\frac{1}{2}\log(N/N_1) + 1, \log N\right]$, we store the $L_p$ and $R_p$ arrays directly using a total of $\mathcal{O}\left(\sum_{p=\lfloor \frac{1}{2}\log(N/N_1)+1 \rfloor}^{\log N} \left(\frac{N}{2^p}(p+1)\right)\right) = \mathcal{O}\left(\sqrt{NN_1} \log N\right)$ bits. Even if we use an MSB table of size $\mathcal{O}\left(N^{1/2} \log\log N\right)$, the total space complexity remains $\mathcal{O}\left(\sqrt{NN_1} \log N \log\left(\frac{1}{\delta}\right)\right)$ bits. Observe that the input array $A$ can be discarded.

*Preprocessing time.* The preprocessing time can be shown to be $\mathcal{O}\left(N + \sqrt{NN_1} \log N \log\left(\frac{1}{\delta}\right)\right)$.

*Query time.* Since extracting the estimated value of each entry of the $L_p$ and $R_p$ arrays takes $\mathcal{O}(t)$ time, the overall query time is also $\mathcal{O}(t) = \mathcal{O}\left(\ln \frac{1}{\delta}\right)$.

*Error bound*

If the query range is larger than $\sqrt{N/N_1}$, the answer is always correct. For smaller queries we use CM sketches. Recall that for $p \in \left[0, \frac{1}{2}\ln(N/N_1)\right]$, we store each $L_p$ (and $R_p$) as a CM sketch with parameter $\varepsilon = \frac{1}{2\times N_1 \times 2^p}$. Hence, the estimated value $\hat{L}_p[i]$ of an entry $L_p[i]$ returned by the CM sketch is between $L_p[i]$ and $L_p[i] + \varepsilon||L_p|| \leq L_p[i] + 0.5$ with probability at least $1-\delta$. In other words, with probability at least $1-\delta$, the CM sketch returns the correct value. In order to answer an R1Q we need to access at most four CM sketches. Hence, with probability at least $(1-\delta)^4 \geq 1-4\delta$, the query will return the correct answer.

We summarize the result.

**Theorem 3.** *Given a 1-D bit array of length $N$ containing $N_1$ nonzero entries, and a parameter $\delta \in \left(0, \frac{1}{4}\right)$, one can construct a data structure occupying $\mathcal{O}\left(\sqrt{NN_1}\log N \log\left(\frac{1}{\delta}\right)\right)$ bits in the encoding model to answer each R1Q correctly in $\mathcal{O}\left(\ln\frac{1}{\delta}\right)$ worst-case time with probability at least $1-4\delta$. For query ranges larger than $\sqrt{N/N_1}$ the query result is always correct.*

By tweaking the algorithm described above slightly, we can reduce the space complexity even further at the cost of providing a weaker correctness guarantee. We assume that we are given an additional parameter $\gamma \in \left(0, \frac{1}{4}\right)$.

For each $p \in [0, \log N]$, we store the $L_p$ and $R_p$ arrays as CM sketches. However, instead of adding a value $v$ directly to a CM sketch, we now add a $(1+\gamma)$ approximation of $v$. More precisely, we add $\lceil \log_{1+\gamma}(1+v)\rceil$ instead of $v$. Hence, for a given $L_p$, the summation of all entries added to its CM sketch is at most $N_1\lceil \log_{1+\gamma}\left(1+2^p\right)\rceil$, and so we set the parameter $\varepsilon$ to $1/\left(2N_1\lceil \log_{1+\gamma}\left(1+2^p\right)\rceil\right)$ for that sketch. The total space used by all CM sketches can be shown to be $\mathcal{O}\left(N_1 \log^3 N \log_{1+\gamma}(1/\delta)\right)$. We store a lookup table of size $\mathcal{O}\left(\log^2 N\right)$ for conversions from $\lceil \log_{1+\gamma}(1+v)\rceil$ to $v$, and an MSB table of size $\mathcal{O}\left(N^{1/c}\log\log N\right)$ for some given integer constant $c > 1$.

We first show that for any given $p \in [0, \log N]$ at most $2\gamma$ fraction of the queries of size $2^p$ can return incorrect answers. Consider any two consecutive blocks of size $2^p$, say, blocks $i \in [0, \frac{N}{2^p} - 1)$ and $i+1$. Exactly $2^p$ different queries of size $2^p$ will cross the boundary between these two blocks. The answer to each of these queries will depend on the estimates of $R_p[i]$ and $L_p[i+1]$ obtained from the CM sketches. Under our construction the estimates are $\hat{R}_p[i] \leq (1+\gamma)R_p[i] \leq R_p[i] + \gamma\cdot 2^p$ and $\hat{L}_p[i+1] \leq (1+\gamma)L_p[i+1] \leq L_p[i+1] + \gamma\cdot 2^p$. Hence, at most $\gamma\cdot 2^p$ of those $2^p$ queries will produce incorrect results due to the error in estimating $R_p[i]$, and at most $\gamma\cdot 2^p$ more because of the error in estimating $L_p[i+1]$. Thus with probability at least $(1-\delta)^2$, at most $2\gamma$ fraction of those $2^p$ queries will return wrong results. Recall from Section 2.1 that we answer given queries by decomposing the query range into two overlapping query ranges. Hence, with probability at least $(1-\delta)^4 \geq 1-4\delta$, at most $2\gamma + 2\gamma = 4\gamma$ fraction of all queries can produce wrong answers.

**Theorem 4.** *Given a 1-D bit array of length $N$ containing $N_1$ nonzero entries, and two parameters $\gamma \in \left(0, \frac{1}{4}\right)$ and $\delta \in \left(0, \frac{1}{4}\right)$, and an integer constant $c > 1$, one can construct a data structure occupying $\mathcal{O}\left(N_1 \log^3 N \log_{1+\gamma}\left(\frac{1}{\delta}\right) + N^{1/c}\log\log N\right)$ bits in the encoding model to answer each R1Q in $\mathcal{O}\left(\log\frac{c}{\delta}\right)$ worst-case time such that with probability at least $1-4\delta$ at most $4\gamma$ fraction of all query results will be wrong.*

*2.4.2. Sampling-based algorithm*

Suppose we are allowed to use only $\mathcal{O}(s)$ bits of space (in addition to the input array $A$), and $s = \Omega(\log_2 N)$. We are also given two constants $\varepsilon \in (0,1)$ and $\delta \in (0,1)$. We build $L_p$ and $R_p$ arrays for each $p \in \left[\log\frac{N}{s} + \log\log N, \log N\right]$, and an MSB lookup table to support constant time MSB queries for integers in $[1, s/\log N]$. Consider the query $\text{R1Q}_A(i,j)$. If $j-i+1 \leq w$, we answer the query correctly in constant time by reading at most 2 words from $A$ and using bit shifts. If $j-i+1 \geq 2^{\log\frac{N}{s}+\log\log N} = \frac{N\log N}{s}$, we use the $L_p$ and $R_p$ arrays to correctly answer the query in constant time. If $w < j-i+1 < \frac{N\log N}{s}$, we sample $\lceil \frac{1}{\varepsilon}\ln\left(\frac{1}{\delta}\right)\rceil$ entries uniformly at random from $A[i\ldots j]$, and return their bitwise OR. It is easy to show that the $L_p$ and $R_p$ tables use $\mathcal{O}(s)$ bits in total, and the MSB table uses $o(s)$ bits of space. The query time is clearly $\mathcal{O}\left(\frac{1}{\varepsilon}\ln\left(\frac{1}{\delta}\right)\right)$.

*Error bound*

If at least an $\varepsilon$ fraction of the entries in $A[i\ldots j]$ are nonzero then the probability that a sample of size $\lceil \frac{1}{\varepsilon}\ln\left(\frac{1}{\delta}\right)\rceil$ chosen uniformly at random from the range will pick at least one nonzero entry is $\geq 1-(1-\varepsilon)^{\frac{1}{\varepsilon}\ln\left(\frac{1}{\delta}\right)} \approx 1-\delta$.

**Theorem 5.** *Given a 1-D bit array of length $N$, a space bound $s = \Omega(\log N)$, and two parameters $\varepsilon \in (0,1)$ and $\delta \in (0,1)$, one can construct a data structure occupying only $\mathcal{O}(s)$ bits of space in the indexing model that in $\mathcal{O}\left(\frac{1}{\varepsilon}\ln\left(\frac{1}{\delta}\right)\right)$ time can answer each*

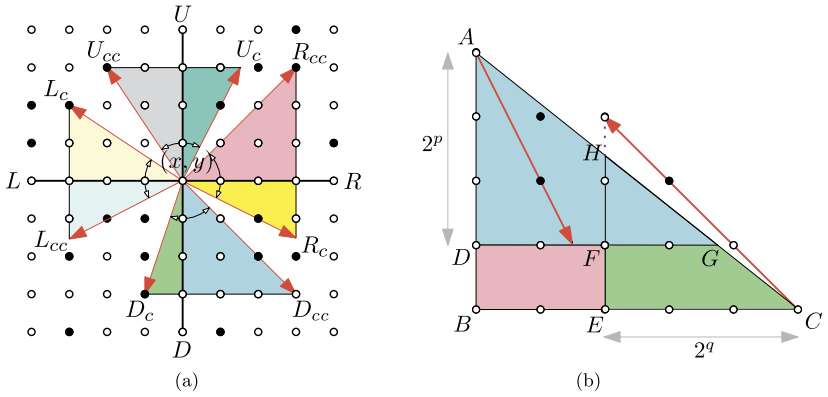


**Fig. 4.** Right triangular R1Q. (*a*) Preprocessing. (*b*) Query execution. Each black grid point contains a 1 while each white grid point contains a 0.

*$R1Q_A(i, j)$ correctly with probability at least $1-\delta$ provided at least an $\varepsilon$ fraction of the entries in $A[i \ldots j]$ are nonzero. If $j-i+1 \leq w$ or $j-i+1 \geq \frac{N \log N}{s}$, the query result is always correct.*

## 3. Non-orthogonal range 1 queries

In this section, we show how to answer R1Q for non-orthogonal ranges, such as axis-parallel right triangles, certain type of polygons, axis-parallel right simplices, and spheres given an input matrix of size $N = n \times n \times \cdots \times n$.

### 3.1. Right triangular R1Q

A right triangular query R1Q(*ABC*) asks if there exists a 1 in an axis-parallel right triangle *ABC* defined by three grid points *A*, *B*, and *C*. In the rest of the paper, right triangles will mean axis-aligned right triangles.

*Preprocessing*

For every grid point $(x, y)$ containing a 0, for each $p \in [0, (\log N)/2]$, we store the coordinates of 8 other grid points for 8 different orientations. For example, consider Fig. 4(*a*) in which each black grid point corresponds to a 1, and each white point corresponds to a 0. In the figure, for the grid point $P = (x, y)$ and $p = 2$, we show the eight black grid points (i.e., $L_C$, $L_{CC}$, $R_C$, $R_{CC}$, $U_C$, $U_{CC}$, $D_C$ and $D_{CC}$ which are present on the arrows) that we store. For example, $U_C$ is a black grid point (present on an arrow) that lies to the right of *PU* such that $PU_C$ makes the smallest angle $\theta_{UC}$ in the clockwise direction with *PU*. The significance of $U_C$ is that no right triangle with a vertical base of length $2^p$ that has one endpoint at *P*, another endpoint to the right of *P*, and whose hypotenuse makes a smaller nonnegative angle than $\theta_{UC}$ in the clockwise direction with the vertical line can contain a 1. The black grid points $U_c, U_{cc}, D_c, D_{cc}$ (present on the arrows) are defined with respect to the angle with the vertical axis of *P* and $L_c, L_{cc}, R_c, R_{cc}$ are defined w.r.t. the horizontal axis of *P*. The subscripts *c* and *cc* represent the angles that are computed in clockwise and anti-clockwise direction, respectively.

*Query execution*

We show how to answer a right triangular R1Q in $\Theta(1)$ time. Say, we want to answer R1Q(*ABC*) (see Fig. 4(*b*)). Let $2^p$ be the largest power of 2 not larger than $|AB|$, and $2^q$ be the largest power of 2 not larger than $|CB|$. Find grid points *D* and *E* on *AB* and *CB*, respectively, such that $|AD| = 2^p$ and $|CE| = 2^q$. Suppose the horizontal line passing through *D* intersects *BC* at *G*, and the vertical line passing through *E* intersects *BC* at *H*. Observe that *G* and *H* are not necessarily grid points. We assume w.l.o.g. that none of the vertices *A*, *B*, and *C* contains a 1 (as otherwise we can answer the query trivially in constant time). Observe that we can answer R1Q(*ABC*) if we can answer R1Q for triangles *ADG* and *CEH*, and the rectangle *BDFE*. R1Q for the rectangle can be answered using our deterministic algorithm described in Section 2.2. R1Q for a right triangle of a particular orientation with height or base length equal to a power of two can be answered in constant time. This is done by checking whether the point stored (from preprocessing) with the appropriate endpoint of the hypotenuse for that specific orientation is inside the triangle or not.

*Complexity analysis*

The input bit matrix is of size $N = n \times n$. The query time is clearly $\Theta(1)$, and the extra space requirement is $\mathcal{O}\left(N \log N + N_0 \log^2 N\right)$ bits, where $N_0$ is the number of 0 bits. We analyze the preprocessing time as follows. For each grid point, we find the distance to the nearest black point in all four directions: top, bottom, left, and right, in $\mathcal{O}(N)$ time using dynamic programming with $\mathcal{O}(N \log N)$ bits of space and use this intermediate data structure for preprocessing. Assume that for each grid point *P*, we have stored information on 8 points in 8 different orientations $\theta$ as explained in

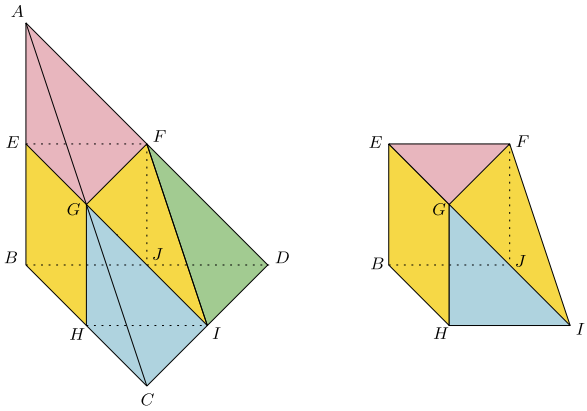


**Fig. 5.** (*a*) An example 3-D simplex. (*b*) Remaining solid when three 3-D simplices are removed from the three corners of the given simplex.

the preprocessing stage, for triangle of base length $2^i$. For simplicity of exposition, we describe here the preprocessing for a single orientation where the stored point, denoted by $S_{PP'}$, is below and to the right of $P$ and the line segment $PP'$ is a vertical line containing $2^i$ grid points. Let $\angle P'PS_{PP'}$ be denoted by $\alpha$. Then, for a vertical line $PP''$ of size $2^{i+1}$ passing through $P'$, $S_{PP''}$ for the same orientation is computed using dynamic programming as follows. For all $2^i$ points between $P'$ (exclusive) and $P''$ (inclusive), the nearest 1-bits to their right are found from the intermediate data structure. Let the angles made by them with the line $PP''$ at the point $P$ be $\alpha_1, \ldots, \alpha_{2^i}$. Then, $S_{PP''}$ is that point that makes the least angle, i.e., $\min(\alpha, \alpha_1, \ldots, \alpha_{2^i})$ with $PP''$. After computing all such stored points, the intermediate data structure can be discarded. Thus, the total preprocessing time is $\mathcal{O}\left(n^2 \cdot \sum_{i=0}^{\lfloor \log n \rfloor}(2^i + 1)\right) = \mathcal{O}\left(n^2 \cdot n\right) = \mathcal{O}\left(N^{1.5}\right)$.

Overall, we obtain the following result.

**Theorem 6.** *Given a 2-D bit matrix of size $N = n \times n$ containing $N_0$ zero bits, one can construct a data structure occupying $\mathcal{O}\left(N \log N + N_0 \log^2 N\right)$ bits in $\mathcal{O}\left(N^{1.5}\right)$ time in the encoding model to answer each axis-aligned right triangular R1Q with the three vertices on the grid points in $\mathcal{O}(1)$ time.*

### *3.2. Right simplicial R1Q*

A $d$-simplex is defined as the generalization of a triangle in $d$ ($\geq 3$) dimensions. We define a right $d$-simplex as a solid obtained by the intersection of the coordinate hyperplanes and an arbitrary hyperplane in $d$-dimensional Cartesian space. Note that as per our definition, a right $d$-simplex is always axis-parallel.

It is not clear how to generalize the right triangular R1Q algorithm described above to answer R1Q in $d$ ($\geq 3$) dimensions. For example, the straightforward generalization of the algorithm does not work for the following reason. A given right triangle (2-D simplex) can be split into two right triangles and a rectangle but a right $d$-simplex ($d \geq 3$) may not be split into $d$ right $d$-simplices with sides powers of 2 and an orthotope. Consider a 3-D simplex *ABCD* as shown in Fig. 5(*a*). Define three new points $E, F, G$ as the midpoints of $AB, AD, AC$, respectively. Remove the three 3-D simplices *AEFG*, *CHIG*, and *DIFJ* from the given simplex *ABCD*. The solid that remains as depicted in Fig. 5(b) is not a cuboid (or an orthotope) and hence we cannot use an orthogonal R1Q algorithm for the solid.

Here, we describe a method to answer right simplicial R1Q for a hypercubic grid of size $N = N^{1/d} \times N^{1/d} \times \cdots \times N^{1/d}$, which works for all dimensions greater than 1.

Preprocessing

Let $P = (x_1, x_2, \ldots, x_d)$ be a grid point. We define grid points $P_1, P_2, \ldots, P_{d-1}$ as $P_1 = (p_1, x_2, \ldots, x_d)$, $P_2 = (x_1, p_2, \ldots, x_d)$, and so on till $P_{d-1} = (x_1, x_2, \ldots, p_{d-1}, x_d)$, where $p_1, p_2, \ldots, p_{d-1} \in [0, N^{1/d} - 1]$. For every grid point $P$ and each value of $p_1, p_2, \ldots, p_{d-1} \in [0, N^{1/d} - 1]$, we store the coordinates of two 1-bits $Q$ and $R$ (if present) satisfying the following. The hyperplane $H_Q$ (resp. $H_R$) passing through $\{P_1, P_2, \ldots, P_{d-1}, Q\}$ (resp. $\{P_1, P_2, \ldots, P_{d-1}, R\}$) minimizes the distance $(P_Q[d] - P[d])$ (resp. $(P[d] - P_R[d])$) while keeping it positive, where $P_Q$ (resp. $P_R$) is the point of intersection of $H_Q$ (resp. $H_R$) and the $d$th coordinate axis; and $P[d]$ represents the $d$th coordinate value of the point $P$.

Query execution

We describe how to answer a right simplicial R1Q in $\Theta\left(d^2\right)$ query time. Given a right $d$-simplex $AB_1B_2 \ldots B_d$, where interior angles at corner $A$ are right angles and point $B_i$ differs from point $A$ only in the $i$th dimension, then we can find R1Q($AB_1B_2 \ldots B_d$) as follows. Given $B_1, \ldots, B_{d-1}$, we store for point $A$ two points $Q$ and $R$ (that differ only in the $d$th coordinate) such that given a grid point $B_d$, (*i*) if $B_d = P_Q$ or $B_d = P_R$, then the query returns a 1, (*ii*) if $B_d$ is neither $P_Q$

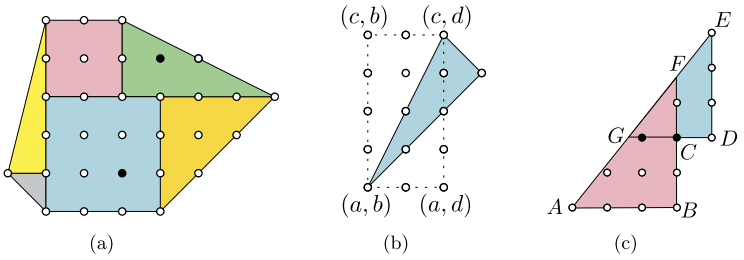


**Fig. 6.** Black grid points contain 1's and white grid points contain 0's. Polygon in (*a*) satisfies Proposition 1. Polygons in (*b*) and (*c*) do not satisfy Proposition 1. Still, R1Q can be answered for (*c*).

nor $P_R$ but lies on the line segment $P_Q P_R$ then the query returns a 0, and (*iii*) if $B_d$ is not on the line segment $P_Q P_R$, then the query returns a 1.

Complexity analysis

The query time is $\Theta\left(d^2\right)$ as there are $d+1$ corners for a $d$-simplex, each grid point has $d$ coordinates. Reading the input and query execution (reading the stored point and computing) both take $\Theta\left(d^2\right)$ for the same reason. The total space is computed as follows. There are $N$ grid points and for every grid point $P$ and $d-1$ points $P_1, P_2, \ldots, P_{d-1}$ (as described in the preprocessing stage), we store two points $Q$ and $R$, which takes $\Theta(\log N)$ bits of space. Hence, the total space required is $\Theta\left(N^{2-1/d} \log N\right)$ bits. The preprocessing time is $\mathcal{O}\left(N^{3-1/d}\right)$ if the preprocessing is done naively. Finding a point $Q$ and $R$ takes $\mathcal{O}(N)$ time. Therefore, the total preprocessing time is $\mathcal{O}\left(N^{2-1/d} \times N\right) = \mathcal{O}\left(N^{3-1/d}\right)$.

Thus, we have the following theorem.

**Theorem 7.** *Given a d-D ($d \geq 2$) hypercubic bit matrix of size $N = N^{1/d} \times N^{1/d} \times \cdots \times N^{1/d}$, one can construct a data structure occupying $\mathcal{O}\left(N^{2-1/d} \log N\right)$ bits in the encoding model in $\mathcal{O}\left(N^{3-1/d}\right)$ time to answer each axis-aligned right simplicial R1Q with $d+1$ vertices on the grid points in $\Theta\left(d^2\right)$ time.*

### 3.3. Polygonal R1Q

Consider a simple polygon with its vertices on grid points satisfying the following property.

**Property 1.** *For every two adjacent vertices $(a, b)$ and $(c, d)$, one of the two right triangles with the third vertex being either $(a, d)$ or $(c, b)$ is completely inside the polygon.*

It can be shown that such a polygon can be decomposed into a set of possibly overlapping right triangles and rectangles with only grid points as vertices that completely covers the polygon (see Fig. 6(*a*)). Examples of polygons that do not satisfy the constraint are given in Fig. 6(*b*, *c*), but we can still answer R1Q for the polygon in Fig. 6(*c*).

**Theorem 8.** *A simple polygon with k vertices satisfying Property 1 can be decomposed into $\mathcal{O}(k)$ right triangles and rectangles and hence the polygonal query can be answered in $\mathcal{O}(k)$ time in the encoding model using the space as given in Theorem 6.*

**Proof.** Every two adjacent vertices, $(a, b)$ and $(c, d)$, of the polygon makes an axis-parallel right triangle with either $(a, d)$ or $(c, b)$ such that the triangle is completely inside the polygon. For a given simple polygon of $k$ vertices, it is easy to see that the number of right triangles that satisfy Property 1 for which R1Q has to be answered is $\mathcal{O}(k)$. When a right triangle with its hypotenuse as one of the polygon edges is removed, it adds atmost two edges, vertical and/or horizontal, to the modified polygon. Removing the $k$ right triangles of the polygon leaves a possibly disjoint set of rectilinear polygons with $\mathcal{O}(k)$ vertices or edges. If we answer R1Q for all such right triangles, we will be left with a disjoint set of rectilinear polygons.

Adding a vertical line passing through a vertex of a rectilinear polygon (see Fig. 7(*a*)) that strictly passes inside the rectilinear polygon splits the rectilinear polygon into two rectilinear polygons with reduced number of vertices. Following the process for other vertices, we end up in a collection of rectangles, for which R1Q can be answered using our deterministic orthogonal algorithms or any other good orthogonal R1Q algorithms.

We need to show that the number of rectangles that are created after we split the rectilinear polygons will be $\mathcal{O}(k)$. We assume that the $\mathcal{O}(k)$ vertices of each rectilinear polygon are sorted with respect to $x$-coordinate. In each rectilinear polygon, for each vertex considered in sorted order, we draw a vertical line passing strictly inside the polygon. This vertical line will touch exactly one horizontal line either above or below the vertex creating a rectangle as shown in Fig. 7(*b*). Each vertex can create at most one other rectangle. Hence, the number of rectangles created after the split will be $\mathcal{O}(k)$.

As the simple polygon with $k$ vertices satisfying Property 1 can be split into $\mathcal{O}(k)$ right triangles and $\mathcal{O}(k)$ rectangles and answering R1Q for each such shape takes constant time, the theorem follows. □

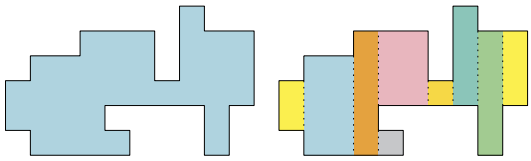

**Fig. 7.** (*a*) A rectilinear polygon. (*b*) A rectilinear polygon with $\mathcal{O}(k)$ vertices can be split into a collection of $\mathcal{O}(k)$ rectangles in $\mathcal{O}(k)$ time.

### 3.4. Spherical R1Q

In this section we present a spherical variant of the R1Q problem. Our result works for all dimensions. We state our result for spheres based on Euclidean distances, but our approach can be used for balls under any distance metric.

The spherical R1Q problem is defined as follows. Given a $d$-dimensional input hypercubic bit matrix $A$ of size $N = N^{1/d} \times N^{1/d} \times \cdots \times N^{1/d}$, preprocess $A$ such that given any grid point $p$ in $A$ and a radius $r \in \mathbb{N}$, find efficiently if there exists a 1 at most distance $r$ away from $p$. Here, we present the algorithm for 2-D. The approach can be extended to $d$ ($\geq 3$) dimensions.

Our method for spherical queries is to, for each point $p$, store the ceiling of the distance to the closest 1, called the ***nearest neighbor***, to $p$. Then a spherical query centered at $p$ does not contain a 1 if and only if the radius $r$ is strictly less than this stored distance. We compress this distance matrix to linear space (in bits) while retaining the $\Theta(d)$ query time.

*Preprocessing*

To preprocess, we begin by computing the distance to the nearest neighbor of each point. We can do this using standard Voronoi diagram techniques: first compute the Voronoi diagram for all 1's in the matrix in $\Theta(N \log N)$ time, then locate each grid point within the diagram in $\Theta(\log N)$ time (see [1] for details). This takes $\Theta(N \log N)$ time in total. We can also find the NN distances of all grid points using multi-source breadth-first search (BFS). Starting from all 1-bits in parallel, we can find the NN distances of their adjacent cells in the next step. Continuing the process using the BFS traversal method, we can compute the NN distances of all cells in $\Theta(N)$ preprocessing time.

We divide the array into $d$-dimensional blocks with all sides of length $((\log N)/4)^{1/d}$. If the array cannot be evenly divided, we pad it with 0's so all blocks are the same size.

Initially, at each point we store the ceiling of the distance to the nearest neighbor. Note that by the triangle inequality, the entry of two neighboring points can only differ by 1.

**Lemma 2.** *Let $p$ and $q$ be adjacent cells on a grid, and $p'$ and $q'$ be the nearest neighbors (closest 1's) to $p$ and $q$ respectively. Let $d(a, b)$ be the distance between two grid points $a$ and $b$. For any distance metric such that $d(p, q) = 1$, then $|\lceil d(p, p') \rceil - \lceil d(q, q') \rceil| \leq 1$.*

**Proof.** Let the distance between two points $a$ and $b$ be denoted by $d_{ab} = d(a, b)$.

If $p' = q'$, we have $d_{pp'} \leq d_{pq} + d_{p'q} = 1 + d_{p'q}$ by the triangle inequality. Similarly, we can find $d_{p'q} \leq 1 + d_{pp'}$. Since both of these are true simultaneously, we get $|d_{pp'} - d_{qq'}| \leq 1$, and then $|\lceil d_{pp'} \rceil - \lceil d_{qq'} \rceil| \leq 1$.

If $p' \neq q'$, without loss of generality, $d_{pp'} < d_{pq'}$ and $d_{qq'} \leq d_{p'q}$. Then $d_{pp'} < d_{pq'} \leq d_{pq} + d_{qq'} = 1 + d_{qq'}$ by the triangle inequality. We also have $d_{qq'} \leq d_{p'q} \leq d_{pq} + d_{pp'} = 1 + d_{pp'}$. To satisfy these simultaneously, $|d_{pp'} - d_{qq'}| \leq 1$, and again the lemma follows. □

We order the points within each block so that subsequent points in the ordering are adjacent. In every block, we store the exact ceiling of the distance to the nearest neighbor[1] for the first grid point. For the remaining points, we store a $+$, $-$, or $=$ symbol if the value is greater than, less than, or equal to the previous point respectively. We call the list of $+$, $-$, or $=$ symbols for each point in a block in order the *type* of the block. An example 2-D block and the symbols used in it is shown in Fig. 8.

By storing distances relative to the previous distance in the ordering, we can achieve $\Theta(N)$ bits of space with $\Theta(d)$ query time. Since each point can only have one of three possible values $(+, -, =)$, and there are $(\log N)/4$ points in each blocks, there can only be $3^{(\log N)/4} \leq \sqrt{N}$ types of block. For each block we only store the type of block, and the ceiling of the distance to the nearest neighbor for the first element. There are $\Theta(N/\log N)$ blocks, so this requires $\Theta(N)$ bits. For each block type, we store for every point in the block the offset between the original distance stored and the distance of the first point. For example, if the first point in some block has a distance 3 to its nearest neighbor, and another point $p$ in the same block has a distance 1 to its nearest neighbor, then an offset of $-2$ will be stored at grid point $p$. The list of offsets is unique for a given block type.

[1] Theorem 7 in the conference version [47] of this paper is incorrect as it assumes that the nearest neighbor of a point $p$ on the interior of a block must either be inside the block, or must also be the nearest neighbor of a point on the boundary of a block, which might not be true for some inputs.

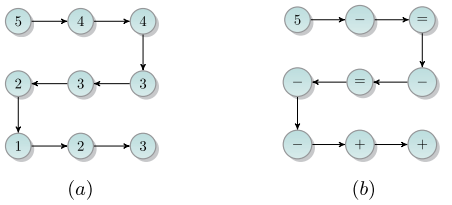


**Fig. 8.** An example 2-D block containing numbers [5 4 4 3 3 2 1 2 3].

*Query execution*

To answer R1Q for a sphere of integer radius $r$ centered at point $p$, we add the distance value stored at the block $b$ containing $p$ to the offset stored in the block type. The sphere does not contain a 1 if and only if the result is strictly greater than the query radius.

It is important to note that our results only apply for integer radii. This allows us to store an integer distance at each point. If we use the squared Euclidean distances to answer spherical queries for real radii, Lemma 2 cannot be used to guarantee that neighboring grid points have a (squared) nearest neighbor distance at most 1 apart. However we still can answer R1Q for some spherical queries centered at a point $p$ for real radius $r'$ in two of the three cases as follows. If the NN distance stored at point $p$ is $(a)$ lesser than or equal to $\lfloor r' \rfloor$, then the sphere contains a 1, $(b)$ greater than $\lceil r' \rceil$, then the sphere does not contain a 1, and $(c)$ exactly equal to $\lceil r' \rceil$, then we cannot answer the R1Q.

The spherical R1Q algorithm presented in this section can be applied to several distance metrics. To extend to arbitrary distance metrics, note that in Lemma 2 we only used the triangle inequality in the proof. Similarly, the ceiling of the NN distance can only change by 1 between adjacent points in any metric. Thus the above techniques extend to any distance metric. Let $p = (p_1, p_2, \ldots, p_d)$ and $q = (q_1, q_2, \ldots, q_d)$ be two points in the $d$-dimensional real vector space. Then, the $L_i$ ($i \in [1, \infty)$ distance between the two points $p$ and $q$, denoted by $||p - q||_i$ is defined as $||p - q||_i = \left(\sum_{j=1}^{d} (q_j - p_j)^i\right)^{1/i}$. For simplicity, consider a 2-D grid. The $L_1$ distance is also called Manhattan distance and the points that have the same Manhattan distance from a point $p$ will be on a rhombus centered at $p$. The $L_2$ distance is also called Euclidean distance and the points that have the same Euclidean distance from a point $p$ will be on a circle centered at $p$. Similarly, the points that have the same $L_\infty$ distance from a point $p$ will be on a square centered at $p$. As all distance metrics from $L_1$ to $L_\infty$ satisfy Lemma 2, we can use the method presented above to answer rhombus R1Q (for $L_1$ distances), hypercube R1Q (for $L_\infty$ distances), and other queries defined by the $L_i$ distances.

*Complexity analysis*

We analyze the above algorithm to prove our bounds. The query time is $\Theta(d)$, as we require a constant number of table lookups and additions.

Finding the NN distances of all grid points takes $\Theta(N)$ preprocessing time using the multi-source BFS traversal method. To add the pointers to each block type, assume we store the block types in an array in lexicographic order. Then the pointer can be computed for each block type using a linear scan. Computing the offsets can also be done using a linear scan, starting at the first element for each block type. For each subsequent element we add 1, subtract 1, or leave the previous offset unchanged for a $+$, $-$, or $=$ respectively. The total time for preprocessing is $\Theta(N)$.

For space, we store $\Theta(N/\log N)$ blocks. For each block we store the block type and the distance to the nearest neighbor of the first element in the block; this takes $\Theta(\log N)$ space. Each block type requires an offset stored for each element, with $\Theta(\log N \log\log N)$ space. Thus the total space is $\Theta\left((N/\log N) \cdot \log N + \sqrt{N} \log N \log\log N\right) = \Theta(N)$ bits.

**Theorem 9.** *Given a $d$-dimensional hypercubic bit matrix of size $N = N^{1/d} \times N^{1/d} \times \cdots \times N^{1/d}$, one can construct a data structure occupying $\Theta(N)$ bits in the encoding model in $\Theta(N)$ time to answer each spherical R1Q for integer radius in $\Theta(d)$ time.*

## 4. Concluding remarks

In this paper, we have presented deterministic and randomized algorithms for orthogonal R1Q having constant query time. The algorithms occupy linear/sublinear space (in bits) in the size of the input matrix. We also presented fast algorithms for non-orthogonal shapes such as axis-parallel right-triangles, certain simple polygons, axis-parallel right simplices, and spheres. We can also answer R1Q for a complex shape if the shape can be expressed as a union of the basic shapes that can be individually answered by R1Q.

A few interesting problems on grids that one could aim to solve in future are

- ▷ How can we efficiently answer R1Q when the bits are changing dynamically?
- ▷ How can we efficiently answer triangular R1Q (any triangle)?
- ▷ How can we efficiently answer R1Q for a polytope (generalization of a polygon in higher dimensions)?

- ▷ What are the tight lower bounds to answer the R1Q problem for different shapes?
- ▷ How can we efficiently find the existence of an element in a given query range (orthogonal and non-orthogonal) in a general matrix?
- ▷ How can we efficiently answer R1Q for complicated shapes in triangular and hexagonal grids?
- ▷ How can we efficiently count the number of 1's for non-orthogonal ranges?
- ▷ How can we efficiently count the number of times an element occurs for orthogonal and non-orthogonal ranges in a general matrix?
- ▷ How can we efficiently solve RMQ and the range partial sum problems for non-orthogonal ranges?

## Acknowledgements

The work of Chowdhury & Ganapathi were supported in part by NSF grants CCF-1162196 and CCF-1439084. Bender & McCauley were supported in part by NSF grants IIS-1247726, CCF-1217708, CCF-1114809, and CCF-0937822. We would like to thank Michael Biro, Dhruv Matani, Joseph S.B. Mitchell, and anonymous referees for insightful comments and suggestions. We thank two anonymous reviewers for giving a tighter bound for the time complexity of the recursive deterministic orthogonal R1Q algorithm and giving an important reference paper [33].

## References

[1] M. De Berg, O. Cheong, M. van Kreveld, M. Overmars, Computational Geometry: Algorithms and Applications, third ed., Springer, 2008.
[2] P.K. Agarwal, J. Erickson, Geometric range searching and its relatives, Contemp. Math. 223 (1999) 1–56.
[3] M. Sharir, H. Shaul, Semialgebraic range reporting and emptiness searching with applications, SIAM J. Comput. 40 (4) (2011) 1045–1074.
[4] J.L. Bentley, J.H. Friedman, Data structures for range searching, ACM Comput. Surv. 11 (4) (1979) 397–409.
[5] P.K. Agarwal, Range searching, Tech. rep., DTIC document, 1996.
[6] J. Fischer, V. Heun, Space-efficient preprocessing schemes for range minimum queries on static arrays, SIAM J. Comput. 40 (2) (2011) 465–492.
[7] M.H. Overmars, Efficient data structures for range searching on a grid, J. Algorithms 9 (2) (1988) 254–275.
[8] G. Navarro, Y. Nekrich, L. Russo, Space-efficient data-analysis queries on grids, Theoret. Comput. Sci. 482 (2012) 60–72.
[9] B. Chazelle, B. Rosenberg, Computing partial sums in multidimensional arrays, in: Symposium on Computational Geometry, 1989, pp. 131–139.
[10] A.C. Yao, Space-time tradeoff for answering range queries, in: Symposium on Theory of Computing, 1982, pp. 128–136.
[11] H. Yuan, M.J. Atallah, Data structures for range minimum queries in multidimensional arrays, in: Symposium on Discrete Algorithms, 2010, pp. 150–160.
[12] M.A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, P. Sumazin, Lowest common ancestors in trees and directed acyclic graphs, J. Algorithms 57 (2) (2005) 75–94.
[13] J. Fischer, V. Heun, H. Stiihler, Practical entropy-bounded schemes for o (1)-range minimum queries, in: Data Compression Conference, 2008, pp. 272–281.
[14] J. Fischer, Optimal succinctness for range minimum queries, in: Latin American Theoretical Informatics Symposium, 2010, pp. 158–169.
[15] M.A. Bender, M. Farach-Colton, The LCA problem revisited, in: Latin American Theoretical Informatics Symposium, 2000, pp. 88–94.
[16] O. Berkman, U. Vishkin, Recursive star-tree parallel data structure, SIAM J. Comput. 22 (2) (1993) 221–242.
[17] A. Amir, J. Fischer, M. Lewenstein, Two-dimensional range minimum queries, in: Combinatorial Pattern Matching, 2007, pp. 286–294.
[18] J. Fischer, V. Heun, A new succinct representation of RMQ-information and improvements in the enhanced suffix array, in: Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, 2007, pp. 459–470.
[19] K. Sadakane, Succinct data structures for flexible text retrieval systems, J. Discrete Algorithms 5 (1) (2007) 12–22.
[20] K. Sadakane, Compressed suffix trees with full functionality, Theory Comput. Syst. 41 (4) (2007) 589–607.
[21] G.S. Brodal, P. Davoodi, S.S. Rao, On space efficient two dimensional range minimum data structures, Algorithmica 63 (4) (2012) 815–830.
[22] A. Golynski, Optimal lower bounds for rank and select indexes, Theoret. Comput. Sci. 387 (3) (2007) 348–359.
[23] R. González, S. Grabowski, V. Mäkinen, G. Navarro, Practical implementation of rank and select queries, in: International Workshop on Experimental Algorithms, 2005, pp. 27–38.
[24] F. Claude, G. Navarro, Practical rank/select queries over arbitrary sequences, in: String Processing and Information Retrieval, 2009, pp. 176–187.
[25] A. Golynski, J.I. Munro, S.S. Rao, Rank/select operations on large alphabets: a tool for text indexing, in: Symposium on Discrete Algorithms, 2006, pp. 368–373.
[26] T.M. Chan, K.G. Larsen, M. Pătraşcu, Orthogonal range searching on the RAM, revisited, in: Symposium on Computational Geometry, 2011, pp. 1–10.
[27] T.M. Chan, M. Pătraşcu, Counting inversions, offline orthogonal range counting, and related problems, in: Symposium on Discrete Algorithms, 2010, pp. 161–173.
[28] B. Chazelle, Lower bounds for orthogonal range searching: I. The reporting case, J. ACM 37 (2) (1990) 200–212.
[29] B. Chazelle, Lower bounds for orthogonal range searching: part II. The arithmetic model, J. ACM 37 (3) (1990) 439–463.
[30] D.R. Clark, J.I. Munro, Efficient suffix trees on secondary storage, in: Symposium on Discrete Algorithms, 1996, pp. 383–391.
[31] J. Barbay, M. He, J.I. Munro, S.S. Rao, Succinct indexes for strings, binary relations and multi-labeled trees, in: Symposium on Discrete Algorithms, 2007, pp. 680–689.
[32] A. Farzan, T. Gagie, G. Navarro, Entropy-bounded representation of point grids, in: Algorithms and Computation, 2010, pp. 327–338.
[33] A. Farzan, T. Gagie, G. Navarro, Entropy-bounded representation of point grids, Comput. Geom. 47 (1) (2014) 1–14.
[34] M. Frigo, V. Strumpen, Cache oblivious stencil computations, in: International Conference for High Performance Computing, Networking, Storage and Analysis, 2005, pp. 361–366.
[35] Y. Tang, R. Chowdhury, B.C. Kuszmaul, C.-K. Luk, C.E. Leiserson, The Pochoir stencil compiler, in: Symposium on Parallelism in Algorithms and Architectures, 2011, pp. 117–128.
[36] Y. Tang, R. Chowdhury, C.-K. Luk, C.E. Leiserson, Coding stencil computations using the Pochoir stencil-specification language, in: USENIX Workshop on Hot Topics in Parallelism, 2011.
[37] G.S. Lueker, A data structure for orthogonal range queries, in: Foundations of Computer Science, 1978, pp. 28–34.
[38] J.L. Bentley, Decomposable searching problems, Inform. Process. Lett. 8 (5) (1979) 244–251.
[39] J. JaJa, C.W. Mortensen, Q. Shi, Space-efficient and fast algorithms for multidimensional dominance reporting and counting, in: Algorithms and Computation, 2005, pp. 558–568.
[40] A. Brodnik, P.B. Miltersen, J.I. Munro, Trans-Dichotomous Algorithms Without Multiplications – Some Upper and Lower Bounds, 1997.

[41] M.L. Fredman, D.E. Willard, Surpassing the information theoretic bound with fusion trees, J. Comput. System Sci. 47 (3) (1993) 424–436.
[42] A. Brodnik, Computation of the least significant set bit, in: Electrotechnical and Computer Science Conference, 1993.
[43] A. Andersson, T. Hagerup, S. Nilsson, R. Raman, Sorting in linear time?, in: Symposium on Theory of Computing, 1995, pp. 427–436.
[44] M.L. Fredman, D.E. Willard, Blasting through the information theoretic barrier with fusion trees, in: Symposium on Theory of Computing, 1990, pp. 1–7.
[45] OEIS sequence A000629, http://oeis.org/A000629.
[46] G. Cormode, S. Muthukrishnan, An improved data stream summary: the count-min sketch and its applications, J. Algorithms 55 (1) (2005) 58–75.
[47] M.A. Bender, R.A. Chowdhury, P. Ganapathi, S. McCauley, Y. Tang, The range 1 query (r1q) problem, in: International Computing and Combinatorics Conference, Springer, 2014, pp. 116–128.