A New Algorithm for Updating and Querying Sub-arrays of Multidimensional Arrays

Pushkar Mishra
Delhi Public School R.K. Puram,
New Delhi - 110022,
India
pushkarmishra@outlook.com

April 2013

Abstract

For the purpose of this paper, updating a sub-array of the given d-dimensional array, with a constant, refers to the addition of the constant to all the elements of the sub-array. Querying a sub-array refers to computation of the sum of all the elements in the sub-array. The 1D updating and querying problem has been studied intensively, and is usually solved using segment trees. This paper presents a new algorithm incorporating Binary Indexed Tree to accomplish the same task. The algorithm is then extended to update and query sub-matrices in a matrix (two-dimensional array). Finally, a general solution is proposed for a d-dimensional array, with N elements along every dimension, having complexity $O(2^d * \log^d N)$ for each operation (query or update). This is an improvement, in terms of time complexity, over previously known algorithms utilizing hierarchical data structures like Quadtree and Octree.

Keywords: Algorithm; Data Structure; Multidimensional Array; Binary Indexed Tree; BIT; Range-Update; Range-Query.

1 Introduction

The problem of updating and querying sub-arrays of multidimensional arrays is of consequence to several fields including image processing, data management and geographical information systems. The Quadtree representations of two-dimensional regions and areas, proposed by Klinger[1] in 1971, has been a field of ongoing research. A 2D space can be represented as a 2D array, with cells denoting unit area. In case of 2D array representation of images, individual cells may denote the pixels. Such a representation of 2D space has several applications. For example, in their paper[2], Samet *et al.* describe the development of a geographical information system which utilizes Quadtrees to store data linked with regions. Consider the case when certain regions of a country have received rainfall, and total rainfall needs to be calculated for the season. Consider another case when different regions in a country have received different amounts of water supply for a month, and the net water supply is to be calculated. Quadtree doesn't support the efficient execution of such queries. The proposed algorithm can therefore be of utility in such cases. It can further be applied to 3D image-processing, in cases when a property common to a range of voxels is to be altered or queried. In the above listed examples, the contribution is on the level of underlying data structure and associated algorithms.

Binary Indexed Tree (BIT) or Fenwick Tree[3] is a data structure mostly used for calculating prefix sums efficiently. For an array of N elements, a BIT allows for calculation of the sum of elements of a range in $O(\log N)$ time. It also allows for the update of elements at single positions

(i.e., adding a constant C to the element at the j^{th} position in the array) in $O(\log N)$ time. Such a set of operations, where individual elements are updated, and range sums are queried, is called *Point-Update Range-Query*.

2 Preliminaries

Let us first give some general definitions for clarity. By $\log n$, we mean $\log_2 n$. An array refers to a 1D array, unless specified otherwise. A sub-array of a d-dimensional array named A, is defined as $[a_1, b_1, c_1, d_1, ... : a_2, b_2, c_2, d_2, ...]$, where a_1, a_2 are coordinates along the 1^{st} dimension, b_1, b_2 are coordinates along the 2^{nd} dimension, and so on. It consists of all the elements $A[a][b][c][d][...]$ for which $a_1 \leq a \leq a_2$, $b_1 \leq b \leq b_2$, $c_1 \leq c \leq c_2$, $d_1 \leq d \leq d_2$, and so on. A 'range' also refers to a sub-array.

$RSUM([a_1, b_1, ... : a_2, b_2, ...])$ refers to the sum of all the elements of the sub-array $[a_1, b_1, ... : a_2, b_2, ...]$.

An update operation on a 1D BIT is depicted as $update(X, C)$ and refers to the addition of constant C at the X^{th} index in the BIT. A query operation is depicted as $query(X)$ and refers to finding the cumulative sum up till X^{th} index in the BIT. The algorithms for update and query operations are the same as the ones in the paper by Peter Fenwick[3, p. 329–333].

An update on a 2D BIT is depicted as $update(X, Y, C)$ and refers to addition of constant C to the Y^{th} element in the X^{th} row. A query is depicted as $query(X, Y, C)$ and refers to the cumulative sum of all elements in $[1, 1 : X, Y]$ of the 2D BIT. The algorithms for update and query operations are carried along each dimension in the manner as the ones for the 1D case.

3 *Range-Update Point-Query* using BIT

Assume that there is a 1D array named A. Each of its values is set to 0. Now, a constant C is to be added to all the elements in the sub-array $[i : j]$. This is done multiple times for arbitrary i, j and C. After some of the update operations, we want to know the value of the $A[k]$, where k is an arbitrary index. This is a case of *Range-Update Point-Query*. A BIT can be made to handle such operations.

Method. Consider the Binary Indexed Tree B associated with array A. Since, in the beginning, all elements in the array are 0, therefore all elements in the BIT are also 0. Whenever we have to add a constant C to all the elements in some sub-array $[i : j]$, two update operations are called on the BIT, which are, $update(i, C)$ and $update(j + 1, -C)$. Now, when $query(k)$ is performed on the BIT, the updated value of the $A[k]$ is returned, and not cumulative sum of elements of A, up till the k^{th} index. This method works on BITs of any number of dimensions.

Proof. After the operation $update(i, C)$, $query(k)$ for $k \geq i$, returns a value which is C more than the value which the operation would have returned if the update wasn't performed. The increment is nullified for all the indices greater than j, by the operation $update(j+1, -C)$. As a result, only those updates affect an index, which include the particular index in their range. Therefore, when query on an index is performed, it returns the sum of all the constants, which were added to the element at that index by the update operations. Since, the array was initially set to 0, sum of all the updates to an element is the value of the element.

4 Problem Definition

Let there be an array A, indexed from 1 to N, with all elements initially set to 0. $A[i]$ refers to the i^{th} element in the array. Q number of operations are to be performed on this array. Each of these operations can be of one the following two types:

i. Given (x, y), add a constant C to all the elements of the sub-array $[x : y]$.

ii. Given (x, y), output the sum of all of the sub-array $[x : y]$.

This problem can further be extended to any number of dimensions. Consider it for a matrix M, having N rows and N columns. $M[i][j]$ refers to the j^{th} element in the i^{th} row. There are Q number of operations to be performed which are of one of the following two types:

i. Given ($x1$, $x2$, $y1$, $y2$), add a constant C to all the elements of the sub-matrix $[x1, y1 : x2, y2]$.

ii. Given ($x1$, $x2$, $y1$, $y2$), output the sum of all the elements of the sub-matrix $[x1, y1 : x2, y2]$.

The above listed operations come under the category of *Range-Update Range-Query*.
The paper will only take up the cases of 1D and 2D array to depict the algorithm being put forward.

5 Previously Known Algorithms

5.1 For 1D arrays

1D version of the problem has conventionally been solved using segment tree with Lazy Propagation technique.[4] Running time of this algorithm is $O(Q * \log N)$, for an array of N elements and Q operations (queries and updates). The space required to execute it is $O(N * \log N)$.[5, p. 226-227]

5.2 For Higher Dimensions

We first prove that Segment trees, generalized to higher dimensions, don't produce an optimal solution.

Proposition 1. A d-dimensional segment tree doesn't support Lazy Propagation technique if $d > 1$.

Proof. Consider updating the sub-matrix $[1, 1 : 4, 3]$ of a 4×4 matrix using a 2D segment tree (a data structure where a segment tree is built for each row of the matrix, and then for each column). Elements in columns 1, 2 and 3 of rows 1, 2, 3 and 4 are to be updated. In lazy propagation technique, the node in the segment tree, which is associated with the range to be updated, is marked, and the update is carried out only when required. Here, the node associated with the rows 1 to 4 can't be marked. This is because not all elements in these rows have to be updated. This example shows that a d-dimensional segment tree doesn't support lazy propagation if $d > 1$.

As a result of *Proposition 1*, the running time for each operation (update or query) on a d-dimensional segment tree ($d > 1$) becomes $O(N^d)$, where N is number of elements along any dimension.

Hierarchical data structures (also known as space-partitioning trees), which work on the basis of regular recursive decomposition of space, are used to solve the case of two or more dimensions.[6]

We take up the example of Quadtree to explain the conventional solution based on hierarchical data structures. A Quadtree[7] recursively divides the given 2D space into four regions. Likewise, a 2D array is decomposed into four squares, which are further decomposed into four squares each, and so on. For simplicity, we assume that the side-length of this 2D array is a power of two. Each node in a Quadtree, built on this array, contains data for a square region.[8] The root node contains data for whole of the 2D array. The leaves of the Quadtree contain data associated with single cells. Once the Quadtree has been built, *Range-Updates* can be done in the same manner as with 1D segment trees (Lazy propagation). Queries also follow the same pattern.

Proposition 2. For an $N \times N$ matrix, the worst case query/update time using a Quadtree is $O(N)$.

Proof. Let us assume a matrix M, with side-length N($= 2^k$). A Quadtree recursively divides M into quadrants. Thus, each node stores information for a square sub-matrix. Now, consider

querying/updating a row in this matrix. No part of this row forms a square region except for the individual elements. Therefore, to update/query this row, we need to visit all the leaves associated with each of its elements. This is because no internal node in the Quadtree contains information specifically for this row, or a part of it. To reach the leaves, the height of the Quadtree has to be travelled. This, addition r steps are required, where r is the height of the tree. It is to be noted that the height of the tree has to be travelled only once. Since, the row contains N elements, thus, the total steps required is $N + r$. Hence, time complexity is $O(N)$. This is the worst-case running time.

In his paper[10, p. 240], Samet states that the worst-case memory requirement for building a Quadtree occurs when region corresponds to checkerboard pattern. We encounter this case when building Quadtree over a 2D array. According to Samet, the number of nodes in such cases is a function of r, where r is height (also known as resolution) of the Quadtree. Since, each node in a Quadtree has 4 nodes, therefore, the total number of nodes in a tree with height r is $\frac{4^r-1}{3}$.[9] Hence, the memory requirement is $O(4^r)$.

For an $N \times N$ matrix, the height of the Quadtree is $\log_2 N + 1$ (height of Quadtree is same as the depth of recursion[9, p. 2–3]). Therefore, the memory requirement is $O(4^{\log_2 N+1})$.

Simplifying the expression:

$$\begin{aligned} 4^{\log_2 N+1} &= 4.4^{\log_2 N} \\ &= 2^2.(2^{\log_2 N})^2 \\ &= 2^2.N^2 = (2N)^2 \end{aligned} \tag{1}$$

A Quadtree can be generalized to any number of dimensions. For example, the 3D version, Octree, divides a 3D array into 8 cubes. This goes on recursively. The worst-case running time can be analyzed on the same lines as in *Proposition 2*, and is $O(N^2)$. Memory requirement is given by $O(8^{\log_2 N+1})$, which simplifies to $O((2N)^3)$.

Therefore, for a Quadtree generalized to d-dimensions, the worst-case running time and memory requirement are $O(N^{d-1})$ and $O((2N)^d)$ respectively.

6 Proposed algorithm

6.1 For 1D arrays

An important observation to be made here, is that if $RSUM([1 : x-1])$ and $RSUM([1 : y])$ can be computed efficiently, then $RSUM([x : y])$ can be calculated in constant time ($RSUM$ has been defined in section 2). This is because,

$$RSUM([x : y]) = RSUM([1 : y]) - RSUM([1 : x-1]) \tag{2}$$

We assume a function $Sum(X)$ which returns the value $RSUM([1 : X])$, and an operation $updateR(x, y, D)$ which adds the value D to each element of the sub-array $[x : y]$.

6.1.1 Update

Let there be an array A, indexed from 1 to 10, with all elements initially set to 0. $Sum(X)$ returns 0 for any index X. We call the update operation $updateR(3, 5, 4)$.
Now, $Sum(X)$ is called for an arbitrary index X. There are three possible cases:

i. $X < 3$

ii. $3 \leq X \leq 5$

iii. $5 < X$

For the first case, the value returned by the function $Sum(X)$ should be 0.
For the second case, the value returned should be 4 if $X = 3$, 8 if $X = 4$ and 12 if $X = 5$.
For the third case, it should be 12.

From the example above, the following inferences can be made:
When an operation $updateR(p, q, D)$ is performed, there is no change in the value of $RSUM([1 : X])$ if $X < p$. If $p \leq X \leq q$, then the value of $RSUM([1 : X])$ changes by $(D * X - D * (p - 1))$. If $X > q$, then $RSUM([1 : X])$ changes by $D * (q - p + 1)$.

Thus, after the operation $updateR(p, q, D)$, the change in the value of $RSUM([1 : X])$ varies with index X. For indices which are less that p, there is no change. For an index X, which is between p and q, the change is given by the function $(D * X - D * (p - 1))$. For all indices $X > q$, the change is given by the function $(0 * X - D * (p - q - 1))$. Since, functions can be added, if we know the sum of all the functions for an index P, the total change to the initial value of $RSUM([1 : P])$ can be calculated by putting P into the net function. To define a linear function at an index, we need to know the coefficient of the variable term X, and the value of the term independent of X (henceforth referred to as independent term), which is to be added.

To add a function to the existing functions at the indices in a range, we require a data structure which allows us to efficiently update the coefficients and the independent terms for over the range. The data structure should also be able to return both the parts of the function at any index, when required.
Thus, we use two 1D BITs and *Range-Update Point-Query* technique to store the functions. Let BIT $B1$ store the coefficient of X and BIT $B2$ store the independent term to be added. When an operation $updateR(p, q, D)$ is called, we need to add the function $(D * X - D * (p - 1))$ to the range $[p : q]$. This means that we need to add D to the coefficient part and $D * (p - 1)$ to the independent term of each index in $[p : q]$. Further, we have to add $-D * (p - q - 1)$ to the independent term of each index greater than q. These additions are done according to the method of *Range-Update Point-Query* in section 2.

6.1.2 Query

Whenever the value of $RSUM([1 : X)]$, for an arbitrary index X, is needed, we call $query(X)$ on $B1$ to get the coefficient and on $B2$ to get constant of the net update function. We multiply the value from obtained from $B1$ with X and add the value obtained from $B2$ to get the value of $RSUM([1 : X])$.

6.1.3 Algorithm

Let there be an array of N elements. Maintain two 1D BITs, $B1$ and $B2$.
For updating a sub-array $[a : b]$ with a constant C:

Step 1. Call $update(a, C)$ and $update(b + 1, -C)$ on $B1$ to update the coefficient part in sub-array $[a : b]$.

Step 2. Call $update(a, C * (a - 1))$ and $update(b + 1, -C * (a - 1))$ to update the independent terms in this sub-array.

Step 3. Call $update(b + 1, -C * (a - b - 1))$ on $B2$, to update the independent terms in $[b + 1 : N]$.

For querying $RSUM([x : y])$:

Step 1. Call $query(x - 1)$ on $B1$ and $B2$.

Step 2. Multiply the value obtained from $B1$ with $(x - 1)$ and add the value obtained from $B2$. Let the result be S_1.

Step 3. Call $query(y)$ on $B1$ and $B2$. Multiply the value obtained from $B1$ with y and add the value obtained from $B2$. Let this value be S_2.

Step 4. The required result is $S_2 - S_1$.

Since, this algorithm performs operations on two 1D BITs, thus, the overall complexity is $(Q * 2 * \log N)$, for Q operations.

6.2 Extension to 2D arrays

It is to be noted that if $RSUM([1,1 : x, y])$ can be computed efficiently, for any x and y, then $RSUM[x_1, y_1 : x_2, y_2]$, for arbitrary x_1, y_1, x_2, y_2, can be calculated easily. This is because,

$$\begin{aligned} RSUM([x_1, y_1 : x_2, y_2]) = & RSUM([1,1 : x_2, y_2]) - RSUM([1,1 : x_2, y_1 - 1]) \\ & - RSUM([1,1 : x_1 - 1, y_2]) + RSUM([1,1 : x_1 - 1, y_1 - 1]) \end{aligned} \tag{3}$$

We assume an operation $SUM(X, Y)$ which returns the value of $RSUM([1,1 : X, Y])$, and an operation $updateS(x_1, y_1, x_2, y_2, C)$ which adds C to each element of the sub-matrix $[x_1, y_1 : x_2, y_2]$.

6.2.1 Update and Query

Let there be an $N \times N$ matrix named M. Throughout this section, we treat M as a 1D array of arrays. This means that M has N elements, and each element is an array of size N. For comprehensibility, we refer to the elements of M as *e-arrays*. Updating a sub-matrix $[x_1, y_1 : x_2, y_2]$ could be understood as updating the range $[y_1 : y_2]$ of all the *e-arrays* in the sub-array $[x_1 : x_2]$ of M. Similarly, $RSUM([x_1, y_1 : x_2, y_2])$ is equal to the sum of values of $RSUM([y_1 : y_2])$ for all *e-arrays* in the sub-array $[x_1 : x_2]$ of M. Thus, a sub-matrix $[x_1, x_2 : y_1 : y_2]$ can be updated or queried by updating or querying the sub-array $[x_1 : x_2]$ of M, using the algorithm devised in section 6.1. The difference is that instead of performing operations on individual elements, *Range-Update Range-Query* operations are to be performed on the sub-array $[y_1 : y_2]$ of *e-arrays*.

According to section 6.1, two BITs are required to handle operations on M (as stated earlier, M is a 1D array of *e-arrays*). These BITs have to be two-dimensional, since, a 1D BIT is needed for each *e-array*. But updating and querying ranges of an *e-array* also requires two 1D BITs. Therefore, four 2D BITs have to be used to handle *Range-Update Range-query* operations on a matrix.

The change in value of $RSUM([1,1 : X, Y])$ can be associated with functions dependent on X and Y.

Proposition 3. The value of $RSUM([1,1 : X, Y])$, for arbitrary X, Y in matrix M, varies according to the function $a_1XY + a_2X + a_3Y + a_4$, where a_1, a_2, a_3 and a_4 are constants.

Proof. M is a 1D array of *e-arrays*. From section 6.1, it is known that the value of $RSUM[1 : X]$ varies linearly with index X. Since, elements of this array are also 1D arrays, from section 6.2, it is evident that the coefficient part of the function at any X varies linearly with Y, where Y is any index along the X^{th} *e-array*. The term independent of X varies linearly with Y too. Therefore, $RSUM([1,1 : X, Y])$ varies according to the function $a_1XY + a_2X + a_3Y + a_4$. This function has four terms. While adding functions, only like terms can be added. Therefore, to maintain and add functions having four terms, for the purpose of *Range-Update Range-Query* on matrices, four 2D BITs are required.

6.2.2 Algorithm

Maintain four 2D BITs $B1$, $B2$, $B3$, $B4$.
To update the sub-matrix $[x_1, x_2 : y_1, y_2]$ of M with C:

Step 1. Call the operation $update(x_1, C)$. Update the sub-array $[y_1 : y_2]$ of those *e-arrays* with C, using the devised in section 6.1, which the update algorithm[3, p. 329–333] visits while executing $update(x_1, C)$.

Step 2. Similarly, call $update(x_2 + 1, -C)$. Use $B1$ and $B2$ to handle *Step 1* and *Step 2*.

Step 3. To update the terms independent of index of M in sub-array $[x_1 : x_2]$, call $update(x_1, C * (x_1 - 1))$ and $update(x_2 + 1, -C * (x_1 - 1))$. Update the sub-array $[y_1 : y_2]$ of the *e-arrays* visited by the update algorithm, with the corresponding constants.

Step 4. Similarly, call $update(x_2 + 1, C * (x_1 - x_2 - 1))$. *Step 3* and *Step 4* is to be handled by $B3$ and $B4$.

Consider an operation $SUM(X, Y)$. To query the value of $RSUM[1, 1 : X, Y]$:

Step 1. Query the value of $RSUM([1 : Y])$ of the X^{th} *e-array* using $B1$ and $B2$ according to the algorithm devised in section 6.1. Multiply this value with X. Let the product be S_1.

Step 2. Query the independent term, which is to be added, by querying the range $[1 : Y]$ of X^{th} *e-array* using $B3$ and $B4$. Name this value S_2.

Step 3. $RSUM([1, 1 : X, Y])$ is given by $S_1 - S_2$.

Since, this algorithm performs operations on four 2D BITs, thus, the overall complexity is $(Q * 4 * \log^2 N)$, for Q operations.

As done in section 6.1, the same algorithm can be reached by figuring out the functions to be added to certain sub-matrices of a matrix, when an arbitrary update operation is called. However, for better understanding of algorithm's extension to arbitrary dimensions, the above method was adopted.

6.3 Generalized to Higher Dimensions

It is evident from the algorithm's extension to matrices that it can be generalized to any number of dimensions. The algorithm for a d-dimensional array can be constructed if the one for $(d - 1)$ dimensions is known.

Proposition 4. For handling *Range-Update Range-Query* operations on a d-dimensional array, 2^d d-dimensional binary indexed trees are required.

Proof. Consider a d-dimensional array T. Assume that 2^K BITs are required to handle operations on array of $(d - 1)$-dimensions. T can be understood as a 1D array, each of whose elements is a $(d - 1)$-dimensional array. Thus, updating and querying the coefficient part at an index X, of the array T, requires 2^K d-dimensional BITs. Similarly, 2^K BITs are required for computing the term independent of X. Thus, the function associated with a d-dimensional array contains twice of 2^K terms. As a result, 2^{K+1} d-dimensional BITs are required. Since, 2^1 BITs are needed for 1D array, by the principle of mathematical induction, it can be said that 2^d are needed for d-dimensional array.

An update or query on a d-dimensional BIT, with N elements along each dimension, requires $O(\log^d N)$ time. By *Proposition 4*, it is known that 2^d such BITs are required to handle a *Range-Update* or a *Range-Query* on a d-dimensional array. Therefore, the overall complexity for each operation becomes $O(2^d * \log^d N)$.

7 Analysis of Running time and Memory

7.1 Running Time

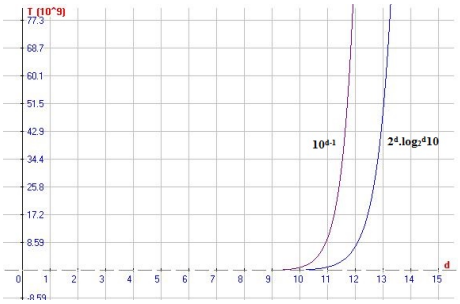
We present the total running time of 100000 update operations (time for query operations shows negligible difference), in milliseconds (ms), of the conventional and the new algorithm for the 2D and the 3D case. The parameters for the update operations, i.e., the sub-matrices to be updated, and the constant for each update were produced using the *rand*() function in the *stdlib.h* of the *C Library*. The data is given for various values of number of elements along each dimension (N):

N	Conventional Algorithm		New Algorithm	
	2D (Quadtree)	3D (Octree)	2D	3D
10	81	522	93	693
50	680	53051	238	1923
100	888	219512	309	3186
150	1387	488673	400	3992
200	3184	910796	431	6357
500	4420		393	
1000	8996		525	
4000	39126		1644	

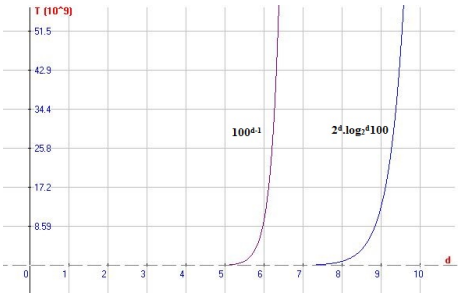
Table 1: Comparison of Running Time

The implementations were done on Ubuntu 13.10 with 8GB RAM and Intel Core i3 3.1 GHz Sandy Bridge processor.

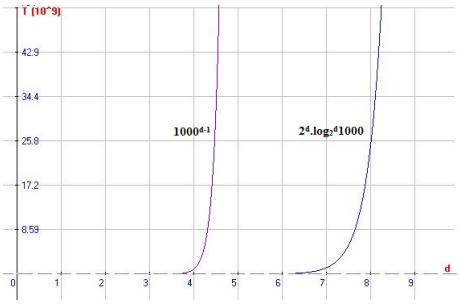
We plot graph of time complexity(T) vs. number of dimensions(d) for an array with 10, 100 and 1000 elements along each dimension.



(a) $N = 10$



(b) $N = 100$



(c) $N = 1000$

Figure 1: Time Complexity vs. Number of Dimensions

7.2 Memory

In his paper, Fenwick states that a BIT requires the same amount of memory as the array on which it is built. For example, for an $N \times N$ matrix, the corresponding 2D BIT will be of the same size. Since, 2^d d-dimensional BITs are required for the proposed algorithm, therefore, the total memory usage is $O((2N)^d)$. This is same as the memory required for executing the conventional algorithm (section 5.2). Therefore, in terms of memory, both algorithms are equally efficient.

8 Conclusion and Future Scope

The algorithm, that this paper puts forward, reduces the time required to update or query sub-arrays of multidimensional arrays significantly. However, the memory needed to execute the algorithm increases exponentially with number of dimensions. We conjecture that an update or query operation on a d-dimensional array can't be done in less than $O(\log^d N)$ time. Therefore, the scope of improvement is only in the amount of memory utilized.

References

[1] A. Klinger. Patterns and search statistics, *Optimizing Methods in Statistics*, J. S. Rustagi, ed., pp. 303-337. Academic Press, New York (1971).

[2] Hanan Samet *et al. A Geographic Information System Using Quadtree.* In *Pattern Recognition*, Volume 17, pages 647–656, 1984.

[3] Peter M. Fenwick. *A New Data Structure for Cumulative Frequency Tables.* In *Software: Practice and Experience*, Volume 24, pages 327–336, 1994.

[4] Lecture. *Segment tree: Design and Application*, section *Lazy Propagation*, pages 18-25. German University in Cairo, 2013.

[5] Mark de Berg, Marc van Kreveld, Mark Overmars, Otfried Schwarzkopf. *Computational Geometry: algorithms and applications (2nd ed.)*. Springer-Verlag Berlin Heidelberg New York, ISBN 3-540-65620-0, 2000.

[6] Hanan Samet. *An Overview of Quadtrees, Octrees, and Related Hierchical Data Structure.* In *Theoretical Foundations of Computer Graphics and CAD*, Volume F40, 1988.

[7] R.A. Finkel, J.L. Bentley. *Quadtrees a data structure for retrieval on composite keys.* In *Acta Informatica*, Volume 4, pages 1–9, 1974.

[8] Pinaki Mazumder. *Planar Decomposition for Quadtree Data Structure.* In *Computer Vision, Graphics, and Image Processing*, Volume 38, pages 258–274, 1987.

[9] Sriram V. Pemmaraju, Clifford A. Shaffer. Department of Computer Science Virginia Polytechnic Institute and State University. *Analysis of the Worst Case Space Complexity of a PR Quadtree.*

[10] Hanan Samet. *Using Quadtrees to Represent Spatial Data.* In *Computer Architectures for Spatially Distributed Data*, Volume F18, pages 229–247, 1985.