IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 3 July 2023, accepted 24 July 2023, date of publication 27 July 2023, date of current version 2 August 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3299352

RESEARCH ARTICLE

Scaled Fenwick Trees

MATTHEW CUSHMAN
Ajna Laboratories, Littleton, CO 80120, USA
e-mail: mcushman@uchicago.edu

ABSTRACT A novel data structure that enables the storage and retrieval of linear array numeric data with logarithmic time complexity updates, range sums, and rescaling is introduced and studied. Computing sums of ranges of arrays of numbers is a common computational problem encountered in data compression, coding, machine learning, computational vision, and finance, among other fields. Efficient data structures enabling $\log n$ updates of the underlying data (including range updates), queries of sums over ranges, and searches for ranges with a given sum have been extensively studied (n being the length of the array). Two solutions to this problem are well-known: Fenwick trees (also known as Binary Indexed Trees) and Segment Trees. The new data structure extends the capabilities for the first time to further enable multiplying (rescaling) ranges of the underlying data by a scalar as well in $\log n$. Scaling by 0 can be enabled, with the effect that subsequent updates may take $(\log n)^2$ time. The new data structure introduced here consists of a pair of interacting Fenwick tree-like structures, one of which holds the unscaled values and the other of which holds the scalars. Experimental results demonstrating performance improvements for the multiplication operation on arrays from a few dozen to over 30 million data points are discussed. This research was done as part of Ajna Labs in the course of developing a decentralized finance protocol. It enables an efficient on-chain encoding and processing of an order book-like data structure used to manage lending, interest, and collateral.

INDEX TERMS Cumulative sums, Fenwick trees, partial sums, prefix sums, segment trees.

I. INTRODUCTION

Consider the problem of storing arrays of numbers so that three distinct operations are efficient

1) Incrementing individual values
2) Calculating cumulative sums over ranges of indices
3) Rescaling values over ranges of indices

Solutions that implement the first two operations in $\log n$ time where n is the length of the list are known. Two of the most commonly used algorithms are Fenwick Trees and Segment Trees. This paper introduces a novel extension of the Fenwick tree that supports the rescaling operation as well in $\log n$ time, called a ''Scaled Fenwick Tree'' (SFT). The idea behind the SFT is to store the underlying values in a traditional Fenwick tree and to encode the scalars in a similar, parallel, Fenwick tree-like data structure, with multiplication instead of addition being the binary operation. The latter Fenwick-like tree encodes the scalar multiples that have been applied to ranges of the data itself. The two trees interact so that incrementing or rescaling a particular value involves traversing both trees in an interactive manner.

Table 1 summarizes the comparative advantages and disadvantages of three implementations for storage and retrieval of arrays: the naive approach (storing in a linear indexed array), Fenwick Trees, and the new method introduced here, Scaled Fenwick Trees. Each method has its strengths and weaknesses, so the best choice for a given application depends on the frequency and circumstances with which one needs to perform each operation. The constants for the Scaled Fenwick Tree are somewhat worse than those for the Fenwick Tree for the simple query/range sum operations and updates. However, the base Fenwick Tree requires super-linear time for range rescaling while the SFT is the only method to offer logarithmic time complexity in that case. Section VI gives experimental data that agrees with the theoretical complexities in Table 1.

Fenwick and Ryabko independently discovered what are now known as Fenwick Trees (or, alternatively, Binary Indexed Trees) in [1] and [2] (also see [3]). For both of these initial papers, the motivation came from dynamic data compression, in which the underlying data were frequency

The associate editor coordinating the review of this manuscript and approving it for publication was Geng-Ming Jiang.

TABLE 1. Comparitive time/space complexity.

	Naive	Fenwick Tree	Scaled Fenwick
Query	O(1)	O(log n)	O(log n)
Range Sum	O(1)	O(log n)	O(log n)
Update	O(n)	O(log n)	O(log n)
Range Multiply	O(n)	O(n log n)	O(log n)
Space	n	n	2n

tables of some tokens in a stream of data. These problems had been studied extensively in the parallel algorithms community [4]. Recently, Bille et al. in [5] consider a data structure storing dynamic partial sums that enables merging adjacent array entries. The theory behind data structures enabling efficient partial summation has been studied as well, with Pătraşcu and Demaine providing tight bounds in [6], and a thorough study of the general structure of storing partial sums by Chaudhuri and Hagerup in [7]. A detailed study of the practical implementation of solutions to the range sum/prefix sum problem, including Fenwick trees and various flavors of Segment trees, is found in [8].

Subsequently, Fenwick trees have found many applications due to their simplicity and efficiency. They are somewhat less general in capability compared to Segment Trees but are more space efficient. Segment Trees are redundant, requiring double space of a naive array or Fenwick tree. Furthermore, because Fenwick trees rely on standard bit operations for indices in twos-complement format, they are easy and efficient to implement on a wide range of architectures. They have found applications in computer vision and graphics (see [9], [10]), statistical regression and kernel estimation (see [11])

Let a_i for $i = 1 \ldots n$ be a sequence of numbers that are to be stored in memory. The most natural way to represent this sequence is to store the raw values sequentially in memory as an array `B[i]` for $i = 1 \ldots n$, so that `B[i]` stores the value a_i. This representation has the merit that changing a value a_i for a particular index i is a constant time operation, as is querying the data structure to determine the value of a_i (we are using an idealized model of computation in which random memory access is constant time). It suffers two drawbacks in addressing problems (1)-(3) however: both (2) and (3) require $\mathcal{O}(n)$ time. To compute the sum of the first k elements, one would need to iterate over all indices up to k, accumulating the sum along the way. A similar process of iterating over the entire array is necessary to scale the first k elements.

One could instead store the values a_i as partial sums, setting `C[i]` to be $\sum_{j=1}^{i} a_i$. In this representation, querying a particular value in the array is also constant time (a_i can be reconstructed as the difference in consecutive values of the array `C`), and computing a prefix sum becomes trivial. However, updating a particular entry a_i becomes expensive: not only must the prefix sum `C[i]` be updated, but also all subsequent values `C[j]` for $j > i$.

In short, there is a distinction between the underlying encoded numerical sequence a_i and the actual representation in memory as a data structure. There are many different ways to encode the sequence a_i, with different time and space trade-offs. In particular, if updating values is rare, but partial sums are often required, representation `C[i]` above might be preferable to `B[i]`. There are data structures that can achieve logarithmic time complexity for both updates and prefix sums, with the trade-off being that simple queries of a particular value themselves also become logarithmic time. One of the most well-studied and efficient was independently discovered by Boris Ryabko and Peter Fenwick ([1], [2]), and is now known as a Fenwick Tree or Binary Index Tree (BIT). The idea behind a Fenwick tree is to find a happy medium between the raw representation `B[i]` and the pure prefix sum representation `C[i]` discussed above. If, instead, certain well-chosen sums of ranges of values in the array are stored, then to update or query a particular index, one only need to access $\mathcal{O}(\log n)$ elements of the array. This enables updating, querying, and computation of partial sums all in logarithmic time.

There are some other operations to consider on our array that are also enabled by Fenwick trees. Searching the array for a particular prefix sum (i.e., finding the largest index i whose prefix sum $\sum_{j=1}^{i} a_i$ is less than a given value) is important for certain applications. This can also be done in logarithmic time using a Fenwick tree. One might also want to consider *range updates*: incrementing some prefix of the tree by a value d, effectively replacing a_j by $a_j + d$ for all $j \leq i$. This can also be done, essentially by modeling the prefix sums as piecewise linear functions of the index, and storing the constants and coefficients in separate Fenwick trees (see [12]).

This paper focuses on the problem of scaling the array values by a given scalar value x. The best-known method for scaling an entire Fenwick tree is to iterate through all of the values in the tree and scale each individually. This linear time operation is actually faster than the most straightforward method of using the Fenwick tree update method to update each value, which is $\mathcal{O}(n \log n)$ (since updates themselves are $\mathcal{O}(\log n)$).

Here, a new data structure that enables logarithmic scaling of any initial segment of the tree by nonzero scalars while preserving logarithmic updates, prefix sums, and searches is introduced. The idea behind the algorithm is to maintain two interacting Fenwick tree-like data structures, one of which (the ''values array'') stores the unscaled values themselves and the other of which (the ''scaling array'') stores the scale factors. The usual invariant of the Fenwick tree, that values in the array store sums of the raw sequence values over particular ranges of indices, is replaced with a new invariant: each entry in the values array times the product of entries encountered in the scale factor array as you traverse from the node towards the root in a particular manner, is equal to the sum of scaled values in a particular range. Scaling by zero can be enabled as well, with a hit to the update operation becoming $\mathcal{O}((\log n)^2)$.

This research was done as part of Ajna Labs in the course of developing a decentralized finance protocol. In this protocol, lenders deposit tokens in an order-book like structure indexed

TABLE 2. Table of terms and definitions.

Term	Definition
n	Array length
a	Sequence of underlying data to be stored, processed and queried
λi	The place of the leftmost nonzero bit of integer i
ρi	The place of the rightmost nonzero bit of integer i
FR[i]	The range of values included in the sum stored in index i of a Fenwick tree. $\{i - 2^{\rho i} + 1, i - 2^{\rho i} + 2, \ldots, i - 1, i\}$
V[i]	The values array of a Scaled Fenwick Tree
S[i]	The scaling array of a Scaled Fenwick Tree
upd(j)	The next index to visit when updating a classic Fenwick Tree; $\text{upd}(j) = j + 2^{\rho j}$
int(j)	The next index to visit when querying a classic Fenwick Tree; $\text{int}(j) = j - 2^{\rho j}$
Upd(j)	The set of iterates of upd applied to j
scale(i)	The total scaling applied to the value at index i of a Scaled Fenwick Tree, equal to $\prod_{j \in \text{Upd}(i)} S[i]$

by price. Computing the amount of deposit above a given price, or finding the price above which a given amount of deposit sits, are both key problems. Furthermore, deposits earn interest, but only if they are priced above a certain level, which was the motivation for the rescaling operation.

II. SYMBOLS AND ABBREVIATIONS

For convenience, Table 2 contains a reference list of commonly used symbols and abbreviations in the text. In all cases, they are also defined or described when introduced.

III. REVIEW OF FENWICK TREES

The following discussion is influenced by Section IV of [13], which has a detailed discussion of the arithmetic relationships between indices that form the basis for Fenwick trees. As Marchini and Vigna discuss, the term Fenwick "tree" is a misnomer, as there is no single tree-like structure relating the indices to one another. Instead, there are three distinct iteration patterns that are used to increment, query, and search through a Fenwick tree. Below is an overview of Fenwick trees to fix the notation.

All arrays and sequences begin with index 1. This is standard in the Fenwick tree literature, as the index calculations become simpler to express in standard bit arithmetic.

For an integer i, define λi be the place of the leftmost (most significant) 1 in the binary expansion of i, and let ρi be its rightmost (least significant) 1. For example, 44 is 101100 in binary, so $\lambda 44 = 5$ and $\rho 44 = 2$.

Let a sequence a_i, which is the underlying data to be stored, encoded as a Fenwick tree V[i]. The principle of the Fenwick tree is to store at index i in the array the sum of the values from the index j with the least significant bit of i cleared, up to index. Define FR(i) (the Fenwick Range of i) to be the these indices:

$$\text{FR}(i) = \left\{ i - 2^{\rho i} + 1, i - 2^{\rho i} + 2, \ldots, i - 1, i \right\} \quad (1)$$

$$\texttt{V[i]} = a_{i-2^{\rho i}+1} + a_{i-2^{\rho i}+2} + \cdots + a_i \quad (2)$$

For example, if i is odd, $\text{FR}(i) = \{i\}$ and $\texttt{V[i]} = a_i$. If i is a power of 2, then $\text{FR}(i) = \{1, 2, \ldots, i\}$ and V[i] is the entire prefix sum up to and including a_i. If $i \equiv 2 \pmod 4$ then $\text{FR}(i) = \{i - 1, i\}$ and $\texttt{V[i]} = a_{i-1} + a_i$.

The following facts are easily verified and are the key observations explaining how Fenwick trees work:

- **FT.1** $i \in \text{FR}(i)$
- **FT.2** For all i, j, $i \neq j$ implies $\text{FR}(i) \neq \text{FR}(j)$. Also, either $\text{FR}(i) \subset \text{FR}(j)$, or $\text{FR}(j) \subset \text{FR}(i)$, or $\text{FR}(i) \cap \text{FR}(j) = \emptyset$
- **FT.3** $j \in \text{FR}(i)$ if and only if i can be obtained from j by iterating the update function $\text{upd}(j) := j + 2^{\rho i}$.
- **FT.4** Let the interrogation function int(j) be the integer obtained by clearing the least significant bit of j's binary expansion: $\text{int}(j) := j - 2^{\rho j}$. The set of positive integers up to and including i is partitioned into the sets FR(j) where j is obtained by iterating int starting at i and ceasing once obtaining 0.

The functions upd and int were introduced in [13] to streamline the discussion of the procedures to *update* and *interrogate* Fenwick trees. In order to increment an underlying value a_i stored in a Fenwick tree (an "update" call) while preserving the invariant 2, one can use property **FT.3**. Increment the value stored in location j of the Fenwick tree itself, V[i] for j being any iterate of the update function upd starting at i. Let Upd(i) be the set of these indices obtained by iterating upd on i. There are only at most $\log n$ such numbers less than n, hence the iteration finished in logarithmic time. Figure 1 illustrates an example of this. The indices are listed in the bottom row of boxes, and the raw underlying data a_i in the row of boxes above that. The Fenwick tree data itself is sorted above, with single solid arrows showing the upd function and dashed arrows the int function. The double solid arrows show the path that the update algorithm would traverse in order to increment the value stored at index 5.

Similarly, using **FT.4**, one obtains the prefix sum of the underlying array by summing V[j] for j being any iterate of int applied to i. There are at most $\log i$ such nonzero iterates. An example is given in Figure 2.

IV. SCALING FENWICK TREES: NONZERO SCALARS

We now move on to the main result of this paper: enabling efficient rescaling of ranges of the underlying data as well. For example, suppose the elements in the underlying array a_i correspond to some statistical observations that fall in particular buckets indexed by i. In order to translate the observations in a probability distribution, one would need to rescale the array by the sum of the entire array to ensure that the sum of values is 1.

The most naive algorithm to do this for data represented in a Fenwick tree would be of order $n \log n$ as follows. Let f be the factor by which the user wants to rescale every element a_i. They could iterate through the array, adding $(f - 1) \cdot a_i$ to the element a_i for $i = 1 \cdots n$. Since the query and update operation is $\mathcal{O}(\log n)$ and they would need to do this n times, the entire process would be $\mathcal{O}(n \log n)$.

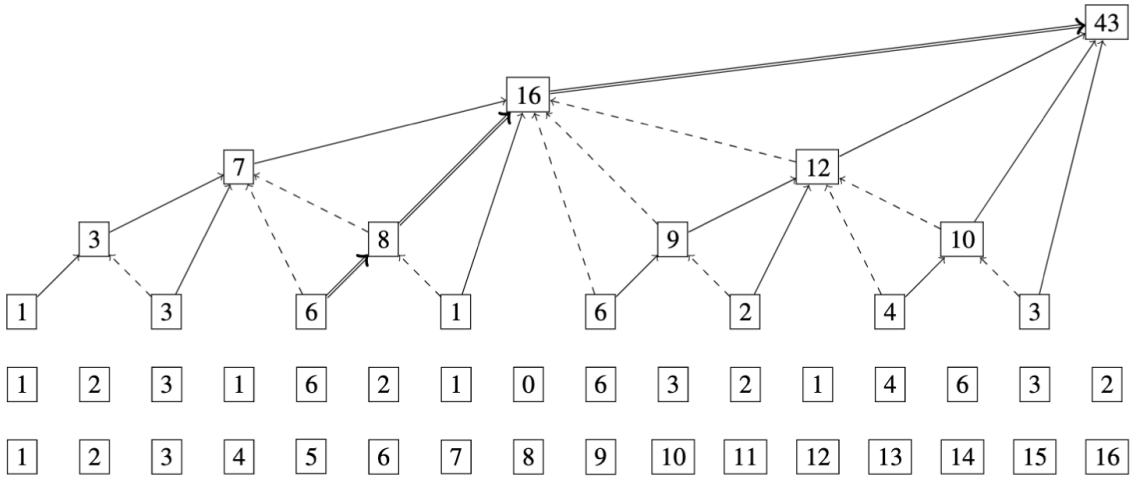

FIGURE 1. Update item 5 by adding 1.

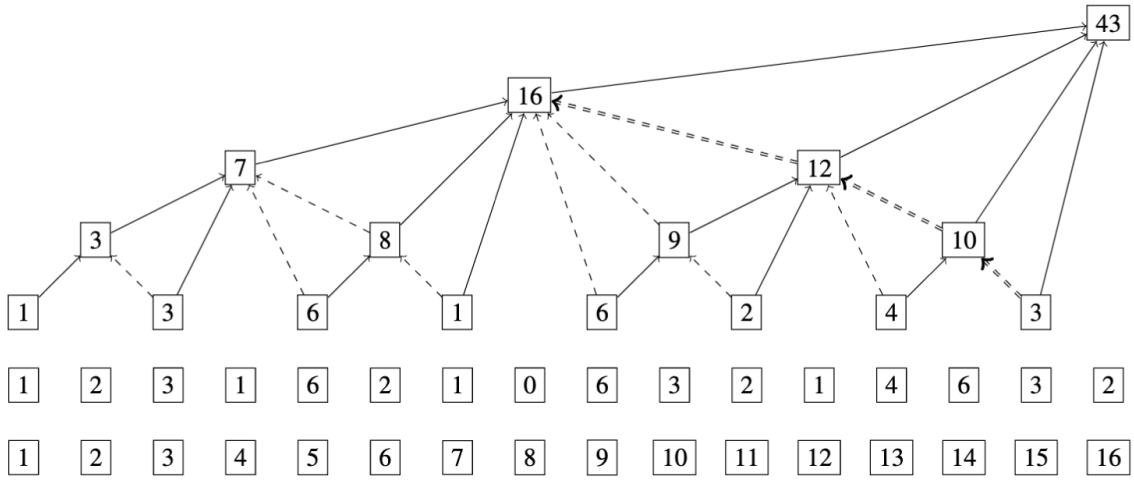

FIGURE 2. Interrogate sum items 1 to 15.

An obvious improvement would be to just iterate directly on the underlying tree itself, scaling each element `V[i]` by the factor f, in $\mathcal{O}(n)$ time.

Starting with any data structure for representing arrays a_i that supports range sums and updates, one can augment the data structure with a global scalar s, which is interpreted as "all elements of array a_i are scaled by s". This would enable global rescaling even more simply and efficiently in $\mathcal{O}(1)$ time. To compute the sum over any desired range, one simply scales the sum by s (relying on the distributive property). To increment or update a_i by a value z, call the increment/update function on the underlying data structure with $s^{-1} \cdot z$, which works fine as long as s is nonzero.

One can generalize this even further by storing the scaling factors themselves in a Fenwick tree-like data structure in parallel to the data that stores the partial sums. Augment the Fenwick tree data `V[i]` with scaling factors `S[i]` for $i = 1 \ldots n$. Intuitively, the value in `S[i]` is regarded as a scaling factor that has been applied to all members of the underlying data a_i for $i \in \mathrm{FR}(j)$. The Fenwick tree invariant (2) that $\texttt{V[i]} = \sum_{j \in \mathrm{FR}(i)} a_j$ is replaced with a more complex invariant involving the scaling factors. Define the scale factor of element i as the product of the factors stored in `S[j]`, as j traverses the Fenwick tree from i to 0 along the same paths used to update the tree:

$$\mathrm{scale}(i) = \prod_{j \in \mathrm{upd}(i)} \texttt{S[j]} \tag{3}$$

We then maintain the invariant:

$$\texttt{S[i]} \star \texttt{V[i]} = \sum_{j \in \mathrm{FR}(i)} a_j \tag{4}$$

The `V[i]` array and `S[i]` array act much like Fenwick trees, but with additional interwoven structure. The `V[i]` are partial sums of the underlying a_j data up to a scalar factor. The `S[i]` encode the scaling factors themselves. The quantity by which one should scale `V[i]` is obtained by starting

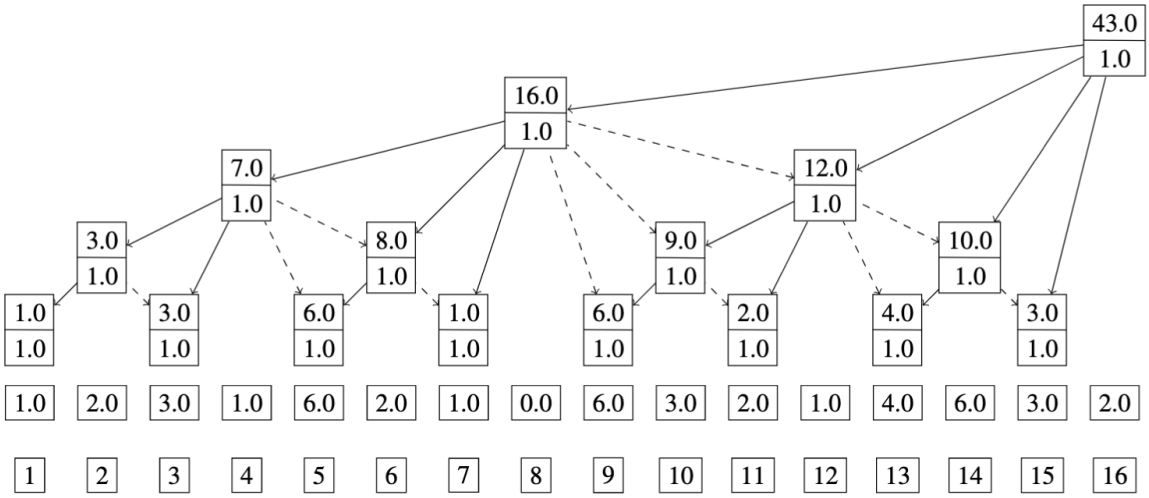

FIGURE 3. Scaled Fenwick tree.

with `S[i]` and iterating up the tree to the root node, accumulating the scaling factors multiplicatively.

Figure 3 shows a representation of a scaled Fenwick tree analogous to the figures presented earlier for standard Fenwick trees. The scaling factors are listed below the values and are initially all set to 1.

Examples:

1) Suppose $n = 2^m$ for some m, and the user wants to compute the prefix sum of values up to 2^k for $k \leq m$. In a standard Fenwick tree, the value stored in `V[2^k]` is the entire prefix sum of the underlying data values up to and including 2^k: `V[2^k]` $= a_1 + a_2 + \cdots + a_{2^k}$, so the value is simply `V[2^k]`. In a scaling Fenwick tree, this value is scaled by the product of the scale factors stored in the S array as traverses along the path up the tree from node to the root, along the increment paths from 2^k to 2^m. These are the powers of 2 between those two indices, so the return value is `S[2^k]*S[2^(k+1)]*...*S[2^m]*V[2^k]`.
2) Again, let $n = 2^m$, and suppose that the users wants to compute the prefix sum of values up to $2^k + 2^j$ for some $j < k \leq m$. In a standard Fenwick tree the value stored in `V[2^k+2^j]` is the sum of values between $2^k + 1$ and $2^k + 2^j$: `V[2^k]` $= a_{2^k+1} + a_{2^k+2} + \cdots + a_{2^k+2^j}$. One can add this to `V[2^k]` to reconstruct the entire prefix sum, so the value returned is `V[2^k]+V[2^k+2^j]`. In a scaling Fenwick tree, these values need to be scaled. As the algorithm proceeds up from $2^k + 2^l$ towards the root, it first passes through the nodes $2^k + 2^{j+1}$, $2^k + 2^{j+1}$ up to $2^k + 2^{k-1}$.

Let the arrays `V[i]` and `S[i]` satisfying invariant (4) be given. One can then construct an entire prefix sum to i in a manner similar to the standard Fenwick tree prefix sum algorithm. As in a standard Fenwick tree, the entire prefix sum is broken into sums over at most $\log n$ subintervals. The new wrinkle is that each of these subinterval sums must be multiplied by the appropriate scale factor as in (4). These scale factors are stored in Fenwick tree structure `S[i]` that parallels the structure of the partial sums `V[i]`, and so they themselves can be accumulated (multiplicatively) alongside the partial sums. Because the factors increase as one moves down the tree, it's more efficient to write this algorithm as moving from the root of the tree down towards the leaves rather than the usual Fenwick tree prefix sum implementation, which iterates from the deeper nodes towards the root.

Below is Python-like pseudo-code implementing the prefix sum algorithm, given tree data `V[i]` and `S[i]`, and index `index`. The variable i traverses the tree downwards from the root towards the target index for the prefix sum. This is accomplished by reconstructing `index` bit by bit, starting with the most significant bit, in contrast with the usual Fenwick algorithm, which starts with `index` and clears it bit by bit, starting with the least significant bit. The new algorithm increments the sum at the same indices as in the standard Fenwick tree prefix sum algorithm, but in reverse order. It also needs to visit some intermediate nodes, however, to track the scale factor itself. For example, in order to compute the prefix sum stored at index 5, the algorithm needs to consider not just the values and scales stored at indices 4 and 5 as in a standard Fenwick tree, but also the scale factor stored at 6 (as well as 8, and any higher power of 2).

In order to increment a specific value in the scaled Fenwick tree at index `index`, one needs only to update the values in Upd(`index`). In the standard Fenwick tree, this is done by traversing the tree upwards using upd starting at `index`, but in this case, how much to increment the value `V[i]` by is unknown, because it's been scaled by scale(*index*) which is the product of all the `S[j]` as `j` traverses the path from `index` to the root along the update path. One could compute this explicitly at the outset, but this would require a redundant traversal. Instead, reversing direction and traversing downwards from the root to `index`

```
1  def prefixSum(values,scales,index):
2      runningSum=0
3      scale=1
4      j=1 << maxNumberOfBitsInIndex
5      i=0
6      while j>0:
7          if index&j:
8              runningSum+=scale*scales[i+j]
9                          *values[i+j]
10         else:
11             scale *= scales[i+j]
12         i=i+(index&j)
13         j= j>>1
14     return runningSum
```

LISTING 1. Prefix sum.

avoids this redundancy. Accumulate the scale factors in `runningScale` along the traversal. Because the value v is added to the value at index i, which is included in the partial sum scaled by `runningScale` at location $ii+j$ in the code below, when incrementing the value array, it is necessary to divide by `runningScale` first.

```
1  def increment(values,scales,index,v):
2      j=1 << maxNumberOfBitsInIndex
3      ii=0
4      runningScale=1
5      while j>0:
6          if (index-1)&j:
7              ii+=j
8          else:
9              runningScale *= scales[ii+j]
10             values[ii+j]+=v/runningScale
11         j = j >> 1
12     return (values, scales)
```

LISTING 2. Increment.

Now consider the algorithm to scale a prefix range of values itself. To scale every entry up to `index` by a number `factor`, one could partition $\{1, \ldots,$ `index`$\}$ into subranges as in **FT.4**. Each one of these subranges can be implicitly scaled by applying the scaling factor to the appropriate entry in the scaling array `S[j]`. This works well for maintaining invariant (4) for `index` itself, but alone would cause the resulting data to violate the same invariant (4) for other indices that overlap but aren't contained in $1 \ldots$ `index`. For example, rescaling the values up to index 5 by only changing the scale factor stored in entries 4 and 5 alone is insufficient: subsequent queries for the sum up to 6 would still reflect the unscaled value at index 5, as this is stored as part of the sum in index 6. The correct algorithm needs to adjust the values stored in these overlapping indices as well.

This can be done by not only traversing upwards through the indices by flipping successive least significant bits to 0 in the binary expansion of `index` as done in `increment`, but by also including intermediate indices that have a single 0 flipped to a 1. The code `mult` below does this by starting with `j` as the least significant bit of `index` and iteratively shifting it left. The variable `runningSum` stores the total increase in sum below `index` in the tree at each loop. If `index` has the same bit set to 1, execute the "if" part (lines 6-8), which scale the subtree below `index` and accumulates in `runningSum` how much they were incremented. Also flip the bit of `index` to 0 as in `increment`. If the corresponding bit in `index` is set to 0, the index is in the overlapping interval case similar to index 6 in the example above. Then increment the corresponding value array element by `runningSum`, and update `runningSum` itself by the corresponding scale factor so that it remains accurate further up the tree.

Figure 4 shows an example of this operating on the SFT presented earlier. The red boxes and blue boxes are the nodes visited when multiplying the 9th entry by 3. Red boxes correspond to the "if" clause, while blue boxes correspond to the "else" clause.

```
1  def mult(values,scales, i, factor):
2      runningSum=0
3      j=i&(-i)
4      while j<=maxIndex:
5          if(i&j):
6              runningSum+=(factor-1)*scales[i]
7                          *values[i]
8              scales[i]*=factor
9              i-=j
10         else:
11             values[i+j]+=runningSum
12             runningSum*=scales[i+j]
13         j = j << 1
```

LISTING 3. Multiply.

Below is the pseudo-code for the inverse prefix sum function as well. This searches the tree for the least index whose sum up to and including it doesn't exceed `target`. This operates very similarly to the analogous search function for standard Fenwick trees, with the addition that it accounts for the scale factor along the way.

V. SCALING FENWICK TREES: ALLOWING FOR ZEROS

The previous section describes a system for scaling ranges of an array of numbers by a nonzero scalar. What happens to this algorithm if passed zero into the `mult` function? This will traverse through certain nodes in the tree, multiplying the entries in the scaling array to 0 – which, of course, merely sets them to the value 0. This implicitly encodes the invariant that effectively says "all values below this in the tree are 0". The only mechanism in the above algorithms to modify the scaling array further is an additional call to `mult`, which can only multiply the entries of the scaling array by subsequent scaling values. Since there is nothing one can multiply 0 by to get a nonzero value, there is no possible way to increment an entry to a nonzero value once it has been set to zero in this way. Another way to see this problem is on line 10 of the pseudo-code for `increment`, which divides by the value in the scaling array. If this value has been set zero, the algorithm will fail with a division by 0.

The issue is this division in `increment`. The existing algorithm does work to scale ranges by zero but breaks subsequent calls to increment values in that range. In the `increment` code, the variable `runningScale` tracks the

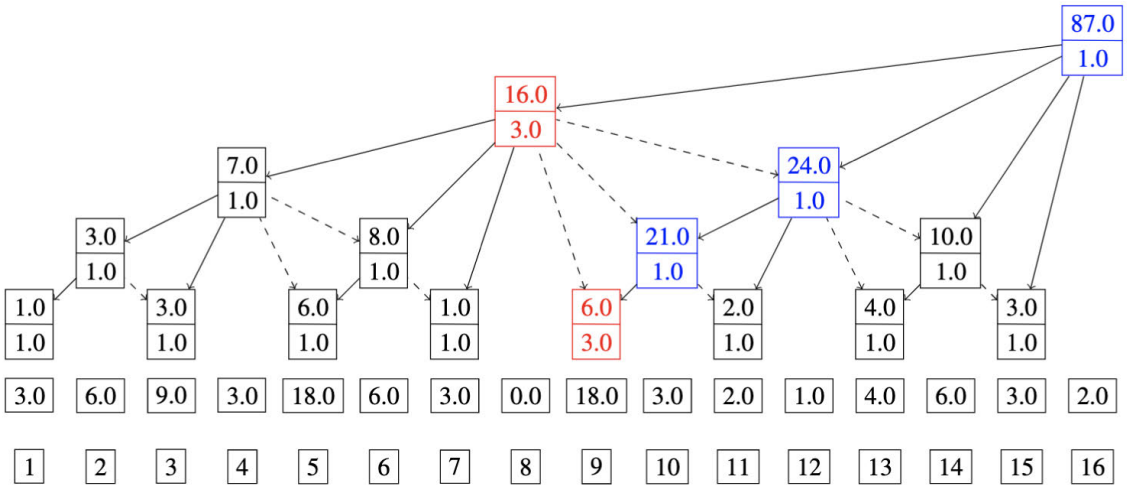

FIGURE 4. Scaling Fenwick Tree Index 9 by 3.

```
def invesePrefixSum(values, scales, target):
    i = 1 << maxNumberOfBitsInIndex
    runningSum = 0
    runningScale = 1
    runningIndex = 0
    while i > 0:
        if runningSum
            +runningScale*values[runningIndex+i]
            *scales[runningIndex+i]
            < target:
            runningIndex+=i
            runningSum+=runningScale
                        *values[runningIndex]
                        *scales[runningIndex]
        else:
            runningScale*=scales[runningIndex+i]
        i=i>>1
    return runningIndex
```

LISTING 4. Inverse query.

```
def incrementAllowingZeros(values, scales, index,
    x):
    j=1 << maxNumberOfBitsInIndex
    ii=0
    runningScale=1
    while j>0:
        if (index-1)&j:
            ii+=j
        else:
            if scales[ii+j]==0:
                scales[ii + j]=1
                values[ii + j]=0
                k=j-1
                while k>0:
                    scales[ii+k]=0
                    k-= k&(-k)
            runningScale*=scales[ii+j]
            values[ii+j]+=x/runningScale
        j = j >> 1
    return (values, scales)
```

LISTING 5. Increment allowing zeroes.

implicit scale factor that is applied to all entries in the `V[i]` array below it in the tree. In order to allow a particular element to be incremented below that, one could reset any 0's encountered in the scaling tree to 1, and then set its value array entry to 0, as well as the scale factor of all of its children to 0. This is simply another way of encoding the data "all entries below this index are 0". This reset will allow the division to proceed.

In order for this to work, $\mathcal{O}(\log n)$ entries in the scaling array need to be set to 0. As the process traverses down the tree, it will continue to encounter these 0, then setting their children to 0 as well. The resulting algorithm has time complexity $\mathcal{O}(\log^2 n)$.

It is possible to modify these algorithms (at least in the nonzero scalar case) to enable the preservation of zeros. Add a new function, `obliterate`, that can force a value to be 0 "on the nose". This addresses the issues with the first example above. One can further modify the other functions that change state, `increment` and `mult`, so that they preserve zero values. The following is a useful alternative characterization of a given index's value "being zero" in an SFT: compute the value at index i by taking the difference between two adjacent prefix sums. These sums are computed by summing the appropriate intervals given in **FT.2**, scaled by the appropriate factors as encoded in the tree. For two adjacent indices $i-1$ and i, many of these intervals (and the associated scale factors) will coincide. The difference comes in index i itself: the prefix sum up to indices including i will include as a term the sum over FR(i) (computed as `scale(i)*V[i]`, while that for $i-1$ will include the sum of over $FR(i-j)$ where j is a power of 2 less than $2^{\rho i}$ (each summand computed as `scale(j)*V[j]`). The scale factors scale(j) that apply to all of the intervals in the latter sum are products of the scale factors encountered as one traverses from index j to the root. These are precisely `S[j]` times the same set of scale factors that appear in the product expansion of scale(i). Therefore, an alternative characterization of an index i having value 0 in the SFT is:

$$\texttt{V[i]} = \sum_{a<\rho i} \texttt{S[i-2\^{}a]*V[i-2\^{}a]} \tag{5}$$

By forcing this equality, one can set the value at index i to 0. Furthermore, if one can always update the tree in such a way that this equality continues to hold after the update if it did prior, then zero values across updates will be preserved across updates.

Equation 5 dictates how to write `obliterate`: just compute the right-hand side, and put it `V[i]`. However, there is one subtlety: after zeroing out the target index, those changes must propagate up the tree to the other nodes that contain i in their range, and do so in such a way that preserves zeros. The key insight is that when modifying a value `V[i]` in the SFT by adding or subtracting a given quantity, this difference can propagate up the tree node by node, computing what would needed to change each node's parent which is then incremented appropriately. The algorithm stores that difference and continues iterating up the tree. As the algorithm traverses the tree upwards to modify node i, it will first have visited one of its children, which will be one of the terms on the right-hand side of (5). Then, modify `V[i]` by `S[i]` times the delta applied child by, which will preserve criterion (5).

Below is pseudo-code for the `obliterate` operation. Lines 2-6 below compute the difference between the left-hand side and right-hand side of (5). Lines 7-11 then apply this difference to the left-hand side of (5) and propagate the difference up the tree.

```
1  def obliterate(values, S, i):
2      j=1
3      runningSum=-values[i]
4      while j&i==0:
5          runningSum+=scales[i-j]*values[i-j]
6          j=j<<1
7      while i<=maxIndex:
8          newValue=values[i]+runningSum
9          runningSum=newValue*scales[i]
10                     -values[i]*scales[i]
11         values[i]=newValue
12         i+=i&(-i)
```

LISTING 6. Prefix sum.

It would be tempting in line 9 of `obliterate` to simply set `runningSum = runningSum*S[i]` – after all, distributing `S[i]` over `V[i]+runningSum` makes this look obvious. However, this would violate the rounding criteria, as the precise change in `V[i]` must be preserved to propagate up the tree further, as discussed above.

Function `increment` needs to change as well and must iterate node by node upwards towards the root to ensure that the differences to every node are propagated.

```
1  def incrementPreservingZeros(values, scales, i, x)
       :
2      x=x/getScale(S, i)
3      while i<= maxIndex:
4          newValue=values[i]+x
5          x=newValue*S[i]-values[i]*scales[i]
6          values[i]=newValue
7          i+=i&(-i)
```

LISTING 7. Prefix sum.

The function `getScale` computes scale(i) by accumulating the product of entries in `S` traversing the int tree from i to the root.

Similar tricks are at play for `mult`. Recall from the discussion of `mult` above that, in addition to applying the new scale factor to various elements of the scaling array `S`, specific overlapping values of the values array `X` also need to be updated as well. Line 4 below computes that delta, and the while loop starting on line 9 applies it consistently to the overlapping intervals.

```
1      def multPreservingZeros(values,scales, i,
       factor):
2          runningSum=0
3          j=i&(-i)
4          while j<=maxIndex:
5              if(i&j):
6                  runningSum+=values[i]*scales[i]*f
7                              -scales[i]*values[i]
8                  scales[i]*=factor
9                  i-=j
10             else:
11                 values[i+j]+=runningSum
12                 runningSum=scales[i+j]*values[i+j]
13                            -scales[i+j]*
14                            (values[i+j]-runningSum)
15             j=j<<1
```

LISTING 8. Multiply.

VI. EXPERIMENTAL RESULTS

Experimental results comparing the time performance of Scaling Fenwick Trees to both a naive implementation and a standard "Base Fenwick" tree implementation of the array interface, including updating, interrogation, and scaling are shown in Figure 5 and Figure 6. The raw numerical values are included in Table 3 and Table 4. The data show a large decrease in scaling times for the Scaling Fenwick Tree. This improvement is offset by small increases for updates as compared to either alternative implementation. For prefix sum queries, the scaled Fenwick tree performs slightly worse than the baseline Fenwick tree, but both tree implementations significantly outperform the naive implementation. This pattern is true both for average and worst-case experimental statistics. All of these empirical results are consistent with the expectations based on the theoretical analysis of the algorithms (namely, that SFTs would offer the best performance for rescaling, with the tradeoff of slightly worse performance for updates and range sum queries).

Python3 code for the Scaled Fenwick Tree and both alternative implementations is available as a reproducible run on Code Ocean in [14]. While the particular experimental results cited here are for a particular desktop machine (which is described below), the results available in [14] agree with these results and can be easily reproduced on Code Ocean.

In considering the reported run-time results, it's essential to bear in mind the specific details of the machines and the implementations used in the experiments. The observed timing, for instance, could be influenced by factors such

TABLE 3. Mean execution time comparison for naive versus baseline fenwick versus scaled fenwick.

	Update Times (ms)			Query Times (ms)			Scale Times (ms)		
$\log_2(n)$	Scaled Fenwick	Naive	Baseline Fenwick	Scaled Fenwick	Naive	Baseline Fenwick	Scaled Fenwick	Naive	Baseline Fenwick
1	0.0156	0.0034	0.0036	0.0033	0.0006	0.0004	0.0152	0.0093	0.0158
3	0.0260	0.0035	0.0038	0.0135	0.0008	0.0005	0.0296	0.0177	0.0353
5	0.0360	0.0034	0.0040	0.0220	0.0022	0.0007	0.0495	0.0516	0.1195
7	0.0429	0.0034	0.0041	0.0299	0.0068	0.0007	0.0623	0.2016	0.4764
9	0.0509	0.0033	0.0043	0.0377	0.0259	0.0009	0.0784	0.7324	1.8006
11	0.0610	0.0034	0.0045	0.0474	0.1152	0.0010	0.0990	2.9509	7.4433
13	0.0701	0.0033	0.0047	0.0567	0.4223	0.0012	0.1168	11.1681	29.1188
15	0.0778	0.0033	0.0049	0.0636	1.5733	0.0013	0.1322	50.0891	134.1686
17	0.0893	0.0034	0.0052	0.0720	6.5255	0.0014	0.1513	176.3270	482.9354
19	0.0974	0.0035	0.0053	0.0811	26.3857	0.0016	0.1654	694.6900	1953.3862
21	0.1043	0.0035	0.0056	0.0914	109.3884	0.0018	0.1844	3166.3422	9177.6498
23	0.1142	0.0035	0.0061	0.0974	347.2394	0.0023	0.2031	11600.0058	34277.0195
25	0.1225	0.0035	0.0067	0.1077	1620.3959	0.0030	0.2158	47479.3940	143150.2115

TABLE 4. Maximum execution time comparison for naive versus baseline fenwick versus scaled fenwick.

	Update Times (ms)			Query Times (ms)			Scale Times (ms)		
$\log_2(n)$	Scaled Fenwick	Naive	Baseline Fenwick	Scaled Fenwick	Naive	Baseline Fenwick	Scaled Fenwick	Naive	Baseline Fenwick
1	0.0378	0.0086	0.0089	0.0055	0.0016	0.0014	0.0184	0.0128	0.0191
3	0.0455	0.0085	0.0041	0.0183	0.0016	0.0010	0.0439	0.0265	0.0531
5	0.0508	0.0037	0.0045	0.0293	0.0039	0.0012	0.0680	0.0921	0.2128
7	0.0657	0.0037	0.0049	0.0364	0.0140	0.0013	0.0813	0.3505	0.8150
9	0.0734	0.0037	0.0049	0.0468	0.0498	0.0013	0.1025	1.3725	3.3697
11	0.0987	0.0037	0.0054	0.0569	0.2016	0.0015	0.1267	5.8937	14.0704
13	0.1074	0.0038	0.0054	0.0706	0.7839	0.0018	0.1505	22.3857	57.4312
15	0.1181	0.0040	0.0059	0.0767	3.2447	0.0020	0.1614	89.5844	238.2480
17	0.1339	0.0040	0.0064	0.0882	12.9848	0.0024	0.1857	364.3869	981.7368
19	0.1533	0.0044	0.0065	0.0934	51.4584	0.0026	0.2038	1443.0576	4019.2751
21	0.1598	0.0048	0.0084	0.1037	204.3815	0.0029	0.2155	5738.4169	16565.4410
23	0.1588	0.0047	0.0086	0.1117	809.4136	0.0043	0.2352	22613.8160	66781.0546
25	0.1691	0.0047	0.0086	0.1276	3298.4628	0.0059	0.2608	91133.6823	271345.1379

as the memory model, including cache utilization and cache coherence. Modern processors make extensive use of caches, and data locality can significantly affect performance. Therefore, an algorithm that makes efficient use of cache can often outperform a theoretically faster algorithm that does not. In the case of the Python implementation, the presence of automatic memory management, or garbage collection, could also impact performance. Garbage collection pauses, often unpredictable, can add significant overhead in terms of time, particularly for programs that create and discard many objects.

Nonetheless, we see consistent behavior across a wide range of array sizes, from a few dozen to ten of millions of indices. The behavior of the average times and the extremal (worst case) times are consistent as well. This provides assurance that these theoretical and empirical results are accurate representations of a typical implementation.

Furthermore, SFTs have been implemented in the Ethereum Virtual Machine using Solidity, a programming language designed for implementing smart contracts running on the Ethereum Virtual Machine, as part of the Ajna Protocol open source project. Run time complexity in Solidity is best measured using gas utilization, and the memory model is very different than a typical x86 based architecture. Despite the entirely different environment and constraints, the results obtained from this implementation were consistent with the theoretical expectations, demonstrating SFT's adaptability

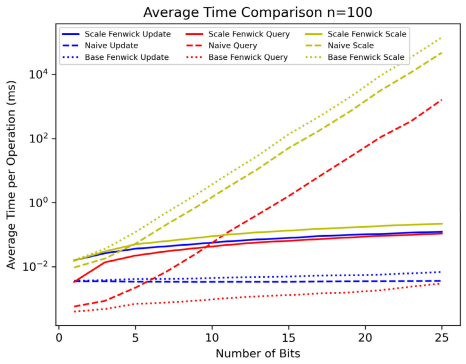

FIGURE 5. Plot of Log Average Execution Time (ms) Versus $\log_2(n)$.

and consistency of the results discussed above. While the particular empirical results discussed in detail here are influenced by many system and implementation factors, the fundamental efficiency of the algorithm as predicted by its theoretical time complexity manifests consistently across diverse platforms.

Figure 5 is a log plot of the average execution time in milliseconds versus array length for all nine pairs of operation "update," "query" and "multiply" with implementation "naive," "baseline Fenwick," "Scaled Fenwick." These were tested using the Python implementation discussed above on an AMD Ryzen 9 5950 3.7 GHz running Ubuntu Linux version 22.04. The x-axis is $\log_2$ of the array length, so the

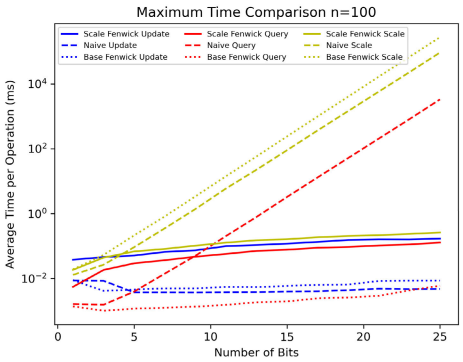

FIGURE 6. Plot of Log Maximum Execution Time (ms) Versus $\log_2(n)$.

longest arrays were of length $2^{25} = 33,554,432$. Each table value is an average of 100 runs for the paired operation and implementation, with the index randomly sampled up to the array length. Identical operations (values and array indices) were used for each of the three different implementations for each operation.

The maximal data among the same 100 runs for each condition are contained in Figure 6. The worst case execution time of the scaled Fenwick tree for these three operations for arrays of length $2^{25} = 33,554,432$ are under a third of a second.

VII. CONCLUSION

Scaled Fenwick Trees are a novel data structure and a suite of algorithms that enable efficient manipulation of numerical array data. Updates, range sums, searches, and multiplying arbitrary ranges of values by nonzero scalars can all be implemented in time logarithmic in the lengths of the array. The data structure is space redundant and requires storing two numerical values for every array entry. This research was motivated by a particular problem in the management of a database of loans and lenders for a blockchain-based decentralized finance application. Similarly, structured problems present themselves in coding and compression, data analysis, filtering and sorting, and other areas, however, so this research may find application well beyond its original motivation. Experimental results show that this algorithm enables sub-second updates, range sums and range rescalings for linear numerical array data of tens of millions of data points on common desktop consumer hardware.

Scaled Fenwick Trees do come with some drawbacks. There is space redundancy in the form of an additional scaling array, so that twice the memory usage is necessary to hold the same number of data points as compared to either a straightforward naive array or classical Fenwick tree. As with classical Fenwick Trees or Segment Trees, updating an array value requires logarithmic, not linear, time in the array length. Finally, compared to a classical Fenwick tree, both updates and range sums require additional computation to incorporate the values in the scaling array so that while both methods are log-time complexity, the constants are worse for the scaled Fenwick tree.

Overall, for efficient implementation of all three operations: updates, range sums and range rescaling of linear array date, Scaled Fenwick Trees offer signficant advantages with reasonable offsetting disadvantages. In applications that require frequent rescalings in particular, Scaled Fenwick Trees can be a good choice of data structure to store and process data.

ACKNOWLEDGMENT

The author would like to thank the valuable conversations with Shiva Chaudhuri, Mike Hathaway, George Niculae, Ed Noepel, Sebastiano Vigna, and Ian Harvey.

REFERENCES

[1] P. M. Fenwick, "A new data structure for cumulative frequency tables," *Softw., Pract. Exper.*, vol. 24, no. 3, pp. 327–336, Mar. 1994.
[2] B. Ryabko, "A fast on-line code," *Sov. Math. Dokl.*, vol. 39, no. 3, pp. 533–537, 1989.
[3] B. Y. Ryabko, "A fast on-line adaptive code," *IEEE Trans. Inf. Theory*, vol. 38, no. 4, pp. 1400–1404, Jul. 1992.
[4] G. E. Blelloch, "Prefix sums and their applications," in *Synthesis of Parallel Algorithms*, J. H. Reif, Ed. 1990.
[5] P. Bille, A. R. Christiansen, P. H. Cording, I. L. Gørtz, F. R. Skjoldjensen, H. W. Vildhøj, and S. Vind, "Dynamic relative compression, dynamic partial sums, and substring concatenation," *Algorithmica*, vol. 80, no. 11, pp. 3207–3224, Nov. 2018.
[6] M. Pătraşcu and E. D. Demaine, "Lower bounds for dynamic connectivity," in *Proc. 26th Annu. ACM Symp. Theory Comput.* New York, NY, USA: Association for Computing Machinery, Jun. 2004, pp. 546–553.
[7] E. W. Mayr, G. Schmidt, and G. Tinhofer, *Prefix Graphs and Their Applications*. Berlin, Germany: Springer, 1995.
[8] G. E. Pibiri and R. Venturini, "Practical trade-offs for the prefix-sum problem," *Softw., Pract. Exper.*, vol. 51, no. 5, pp. 921–949, May 2021.
[9] C. Reinbold and R. Westermann, "Parameterized splitting of summed volume tables," *Comput. Graph. Forum*, vol. 40, no. 3, pp. 123–134, Jun. 2021.
[10] J. Schneider and P. Rautek, "A versatile and efficient GPU data structure for spatial indexing," *IEEE Trans. Vis. Comput. Graph.*, vol. 23, no. 1, pp. 911–920, Jan. 2017.
[11] Y. Wang, Y. Wu, and S. S. Du, "Near-linear time local polynomial nonparametric estimation with box kernels," *INFORMS J. Comput.*, vol. 33, no. 4, pp. 1339–1353, Feb. 2021.
[12] P. Mishra, "A new algorithm for updating and querying sub-arrays of multidimensional arrays," 2013, *arXiv:1311.6093*.
[13] S. Marchini and S. Vigna, "Compact Fenwick trees for dynamic ranking and selection," *Softw., Pract. Exper.*, vol. 50, no. 7, pp. 1184–1202, Jul. 2020.
[14] M. Cushman. (2023). *Scaled Fenwick Tree Reference Implementation, Validation, Time Benchmarks and Comparisons*. [Online]. Available: https://www.codeocean.com/

MATTHEW CUSHMAN received the B.S. degree in mathematics and logic and computation and the M.S. degree in mathematics from Carnegie Mellon University, Pittsburgh, PA, USA, and the Ph.D. degree in mathematics from The University of Chicago. He was the Managing Director of the Knight Capital Group, from 2002 to 2011, the Senior Managing Director of Citadel Securities, from 2011 to 2013, and the Co-Founder of Engineers Gate, in 2014. He left Engineers Gate, in 2017, to found Etale Inc., a trading software firm that was acquired by NYDIG, in 2020. Since January 2022, he has been the Co-Founder of Ajna Laboratories, where he works on smart contract protocol design and implementation.