Succinct Data Structures for Searchable Partial Sums *

Wing-Kai Hon[1], Kunihiko Sadakane[2], and Wing-Kin Sung[3]

[1] Department of Computer Science and Informations Systems,
The University of Hong Kong, Hong Kong,
{wkhon,twlam}@csis.hku.hk
[2] Department of Computer Science and Communication Engineering,
Kyushu University, Japan,
sada@csce.kyushu-u.ac.jp
[3] School of Computing, National University of Singapore, Singapore,
ksung@comp.nus.edu.sg

Abstract. The *Searchable Partial Sums* is a data structure problem that maintains a sequence of n non-negative k-bit integers; in addition, it allows us to modify the entries by the *update* operation, while supporting two types of queries: *sum* and *search*. Recently, researchers focus on the succinct representation of the data structure in $kn + o(kn)$ bits. They study the tradeoff in time between the query and the update operations, under the word RAM with word size $O(\lg U)$ bits. For the special case where $k = 1$ (which is known as *Dynamic Bit Vector* problem), Raman et al. showed that both queries can be supported in $O(\log_b n)$ time, while *update* requires $O(b)$ amortized time for any b with $\lg n / \lg\lg n \leq b \leq n$. This paper generalizes the study and shows that even for $k = O(\lg\lg n)$, both query and update operations can be maintained using the same time complexities. Also, the time for *update* becomes worst-case time.
For the general case when $k = O(\lg U)$, we show a lower bound of $\Omega\left(\sqrt{\lg n / \lg\lg n}\right)$ time for the *search* query. On the other hand, we propose a data structure that supports *sum* in $O(\log_b n)$ time, *search* in $O(\tau \log_b n)$ time, and *update* in $O(b)$ time, where τ denotes the value of $\min\left\{\lg\lg n \lg\lg U / \lg\lg\lg U, \sqrt{\lg n / \lg\lg n}\right\}$. When $b = n^{\epsilon}$, our data structure achieves optimal time bounds.
This paper also extends the *Searchable Partial Sums* with *insert* and *delete* operations, and provides succinct data structure for some cases.

1 Introduction

The size of electronic data is growing exponentially, whose rate is now exceeding that of computer memory. To battle the information avalanche, the quest for highly space-efficient or *succinct* data structures, which are asymptotically optimal in operation times, is desirable, and has attracted many researchers recently [4, 7, 8] (See [7] for more references). In this paper, we revisit a well-studied data structure problem called *Searchable Partial Sums*, and provide succinct solutions to it. We assume a unit-cost word RAM with word size $O(\lg U)$ bits,[4] in which

* This work was supported in part by the Hong Kong RGC Grant HKU-7024/01E; by the Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan; by the NUS Academic Research Grant R-252-000-119-112.

[4] We use the notation $\lg n$ and $\log_b n$ to denote the base-2 and base-b logarithm of n, respectively.

standard arithmetic and bitwise boolean operations on word-sized operands can be performed in constant time [1, 5].

Let $A[1], A[2], \ldots, A[n]$ be a sequence of n non-negative k-bit numbers with $n \leq U$. The *Searchable Partial Sums* problem maintains the sequence under two query operations `sum` and `search`, and an `update` operation as follows:

- `sum`(i): return the value $\sum_{t=1}^{i} A[t]$.
- `search`(j): return the smallest i such that `sum`$(i) \geq j$.
- `update`(i, δ): set $A[i] \leftarrow A[i] + \delta$, for some integer δ such that $0 \leq A[i] + \delta \leq 2^k - 1$ and $-\lg^{O(1)} n \leq \delta \leq \lg^{O(1)} n$.

When $k = 1$, such special case is commonly known as the *Dynamic Bit Vector* problem, which maintains a bit vector of length n under two query operations `rank` and `select`, and an update operation `flip`. These operations are essentially the `sum`, `search` and `update`, respectively. In what follows, we adopt the above common notations when dealing with bit vectors.

The *Searchable Partial Sums* was originated from a related problem called *Partial Sums*, which does not consider the `search` query. Fredman and Saks [3] showed that, in the *cell probe* model of computation (Yao [9]), [5] an intermixed sequence of n queries and updates requires $\Omega(\lg n / \lg\lg n)$ amortized time per operation. Dietz [2] proposed a data structure of $\Theta(n \lg n)$ bits for the *Partial Sums* that supports `sum` and `update` optimally in $O(\lg n / \lg\lg n)$ time, for $k = \Theta(\lg n)$. Raman, Raman and Rao [7] generalized the result to $k = O(\lg U)$, and reduced the space required to $kn + o(kn)$ bits. This space is information-theoretically optimal within a lower-order term. In particular, they gave a tradeoff result that supports `sum` in $O(\log_b n)$ time and `update` in $O(b)$ time, for any parameter $b \geq \lg n / \lg\lg n$. Note that when $b = \lg n / \lg\lg n$, all operations take $O(\lg n / \lg\lg n)$ time.

For *Searchable Partial Sums*, Raman, Raman and Rao [7] gave a succinct structure that supports all operations in $O(\lg n / \lg\lg n)$ worst-case time. Concerning tradeoff, they can only give a succinct solution with the same tradeoff between query and update times for $k = 1$. (That is, $O(\log_b n)$ time for `rank` (`sum`) and `select` (`search`), while $O(b)$ time for `flip` (`update`).) Moreover, their update time becomes amortized instead of worst-case. There are two open questions: 1. *Can we improve the amortized time to worst-case time for $k = 1$?* 2. *Can we achieve the same tradeoff for $k > 1$?*

In this paper, we investigate the *Searchable Partial Sums* for different ranges of k, and propose corresponding succinct structures that allow tradeoff between query and update times. The space of the structures are all $kn + o(kn)$ bits. For $k = 1$, we can support `sum` and `search` in $O(\log_b n)$ time, while `update` in worst-case $O(b)$ time. This improves the previous result which requires amortized update time, and answers affirmatively to the first open question. Furthermore, we extend the same tradeoff for $k = O(\lg\lg n)$, thus answering partly to the second open question.

For the general case when $k = O(\lg U)$, we give a negative example for the second open question, by showing a lower bound of $\Omega\left(\sqrt{\lg n / \lg\lg n}\right)$ time for `search`. On the other hand, our proposed structure can support `sum` in $O(\log_b n)$ time, `update` in $O(b)$ time and `search` in $O(\tau \log_b n)$ time, where $\tau = \min\left\{\lg\lg n \lg\lg U / \lg\lg\lg U, \sqrt{\lg n / \lg\lg n}\right\}$. Thus, our data structure achieves optimal time bounds when $b = n^\epsilon$ for any fixed $\epsilon > 0$.

[5] In the cell probe model, the time complexity of a sequential computation is defined to be the number of words of memory that are accessed. The lower bound time complexity derived is stronger than that in the word RAM model.

Technically speaking, when $k = 1$, our improvement stems from a better solution for the `select` query. Existing method [7] maintains a weight-balanced B-tree (WBB tree) of Dietz [2], and updating of which can only be done in amortized time. In this paper, we modified the WBB tree so that the relevant information in the internal nodes for efficient query can be stored compactly, and updated efficiently. Consequently, the `select` time is maintained, while the `flip` time becomes worst-case time.

When $k > 1$, we observe that `search` can be supported efficiently by coupling an efficient WBB tree with our new solution to the *Dynamic Bit Vector* problem. The idea is to divide the input array into fixed-sized groups, and use a bit vector to remember those groups whose sum of elements is positive. With `select`, we can access any *positive-sum* groups easily. Then, we can focus on those *positive-sum* groups, and use our WBB tree to manage them.

We also consider the *Searchable Partial Sums with Indel* problem. This problem extends the *Searchable Partial Sums* problem to consider a resizeable sequence, which supports `insert` and `delete` operations as follows:

- `insert`(i, x): insert a new integer x between $A[i]$ and $A[i+1]$.
- `delete`(i): remove the entry $A[i]$ from the sequence.

When the sequence consists of only positive k-bit integers and $k = O(1)$, we provide a $kn + o(kn)$-bit data structure that supports `sum` and `search` in $O(\log_b n)$ time, `update` in $O(b)$ time, while `insert` and `delete` in amortized $O(b)$ time for $b \geq \lg^{O(1)} U$, where n is the current size of the sequence. On the other hand, if we are given an extra $O(U^\epsilon)$-bit precomputed table for any fixed $\epsilon > 0$, we can extend k to $O(\lg\lg U)$, and all the three update operations can be supported in worst-case $O(b)$ time.

The paper is organized as follows. The next section introduces some basic lemmas for building our data structures. Section 3 discusses the data structure for *Searchable Partial Sums with Indel*. In Sections 4, 5, and 6, we investigate the *Searchable Partial Sums* for $k = 1$, $O(\lg\lg n)$ and $O(\lg U)$, respectively. Concluding remarks are given in Section 7.

2 Preliminaries

The first two lemmas, which are rephrased from [7], form part of the results in the paper.

Lemma 1. ([7], Corollary 2) *Given a bit vector of length n. Together with an auxiliary data structure of $o(n)$ bits of space, we can support* `rank` *in $O(\log_b n)$ time and* `flip` *in $O(b)$ time, for any parameter b with $\lg n / \lg\lg n \leq b \leq n$.*

Lemma 2. ([7], Theorem 2 with minor adaptation) *Given a sequence of n non-negative k-bit integers that is stored in an array, where $k = O(\lg U)$. Together with an auxiliary data structure of $o(kn)$ bits of space, we can support* `sum` *and* `search` *in $O(\log_b n)$ time and* `update` *in $O(b)$ time for any parameter b with $\lg n / \lg\lg n \leq b \leq \lg^{O(1)} n$.*

Then, the following lemmas provide fundamental tools for our data structure.

Lemma 3. *Given a sequence G of non-negative k-bit integers that is stored in an array, where $|G| = \lg^{O(1)} L$ and $k = O(\lg\lg L)$ for some $L \leq U$. Together with an auxiliary data structure of $o(|G|)$ bits of space and a precomputed table of size $O(L^\epsilon)$ bits for any fixed $\epsilon > 0$, we can support* `sum`*,* `search` *and* `update` *in $O(1)$ time. Moreover, the auxiliary data structure can be constructed in $O(|G| \lg\lg L / \lg L)$ time.*

Proof. Omitted (deferred to the full paper). □

Lemma 4. ([7], Lemma 2) *Given a sequence of n non-negative k-bit integers, where $k = O(\lg U)$. There is a data structure of size $O(kn)$ bits that supports* `sum`, `search` *and* `update` *in $O(\lg n / \lg \lg n)$ time.*

Lemma 5. (Rephrased from [6]) *Given a bit vector of length n. Together with an auxiliary data structure of $o(n)$ bits of space, we can support* `rank` *in $O(1)$ time. The construction of the auxiliary data structure takes $o(n)$ time.*

3 Searchable Partial Sums with Indel

This section discusses the problem of *Searchable Partial Sums with Indel*. Firstly, we have the following lemma.

Lemma 6. *Given a sequence of at most L integers $p_1, p_2, p_3, \ldots$, with $x/2 \leq p_i \leq x$ for some x. Then, there is a data structure of size $O(z \lg(bxL))$ bits that supports* `sum` *and* `search` *in $O(\max\{\log_b z, 1\})$ time, and* `update`, `insert` *and* `delete` *in $O(b)$ time, for all $b \geq (\log_b z)^2$ where z denotes the current size of the sequence.*

To prove the above lemma, we maintain a weight-balanced B-tree (WBB tree) over the z integers as follows. Each integer represents a leaf, and all the leaves are at the same depth. Define *level* of a node to be the depth of the tree minus the depth of the node, and define *weight* of a node to be the total sum of integers under the node. Each internal node at level i maintains an invariant that its weight is at least $b^i x/2$ and at most $b^i x$.

For each internal node at level i, we store:

- the LW-value, which is the sum of integers under all its left siblings. (Note: the LW-value is stored using $(i+1)\lg(bx)$ bits.)
- the LN-value, which is the number of integers under all its left siblings. (Note: the LN-value is stored using $(i+1)\lg(b)$ bits.)
- an auxiliary data structure $\mathcal{D}_1$ that returns the leftmost child such that its LW-value is at least j, for any j, in $O(1)$ time. The data structure occupies $O(b)$ bits, and can be constructed in $O(b)$ time.
- an auxiliary data structure $\mathcal{D}_2$ that returns the leftmost child such that its LN-value is at least j, for any j, in $O(1)$ time. The data structure occupies $O(b)$ bits, and can be constructed in $O(b)$ time.
- pointers to its children, each using $O(\lg L)$ bits.

For each leaf node, we simply store the LW-value and the LN-value. Both of them are stored using $\lg(bx)$ bits.

Suppose the auxiliary data structures $\mathcal{D}_1$ and $\mathcal{D}_2$ work with the claimed bounds. Immediately, we have the following propositions.

Proposition 1. *The space occupied by the above B-tree is $O(z \lg(bxL))$ bits.*

Proof. (sketch.) By straightforward calculation. □

Proposition 2. *It takes $O(\log_b z)$ time to perform* `search` *and* `sum` *operations by the above WBB tree.*

Proof. (sketch.) To support `search`, we start from the root and descend level by level to the leaf $p_{\texttt{search}(j)}$ based on the LW-values and $\mathcal{D}_1$ of the traversed nodes. The required answer is the sum of LN-values on the traversal path. Time bound follows as descending each node takes $O(1)$ time and the height of the tree is $O(\log_b n)$. Similar arguments for the `sum` operation. □

Proposition 3. *It takes $O(b)$ time to support* `update`, `insert` *or* `delete` *in the above WBB tree.*

Proof. (sketch.) To support the update operations, we follow standard procedure for updating a WBB tree. A straightforward calculation yields an update time of $O(b \log_b z)$. Then, we set b to $b^{1/2}$, so that update time becomes $O(b)$ while the other propositions remain correct. □

Now, it remains to describe the auxiliary data structures $\mathcal{D}_1$ and $\mathcal{D}_2$ in the internal node, as shown in the following proposition.

Proposition 4. *Auxiliary data structures $\mathcal{D}_1$ and $\mathcal{D}_2$ can work with the claimed bounds.*

Proof. To implement $\mathcal{D}_1$ for each internal node, one way is to use a bit vector such that the i-th entry is marked as `1`, if one of LW-values of its children is i. Together with an auxiliary data structure of Lemma 5 that supports $O(1)$-time `rank` operation on this bit vector, the required query on $\mathcal{D}_1$ can be answered by a `rank` query.

Observe that the sum of integers under each level-i node is at least $b^i x/2$ and at most $b^i x$. A brute force approach would require $b^i x(1 + o(1))$ bits for an internal node at level i, which is too spacious. However, the observation also implies the sum under each child of a level-i node is at least $b^{i-1}x/2$, so that when the LW-values of all these children are divided by $b^{i-1}x/2$, each gives a distinct value of at most $2b$. In other words, the LW-values of the children (after dividing by $b^{i-1}x/2$) can be stored in $2b(1+o(1))$ bits, while we still can answer the required query on $\mathcal{D}_1$ in $O(1)$ time. Moreover, the construction of this data structure takes $O(b)$ time. The auxiliary data structure $\mathcal{D}_2$ can be constructed and stored similarly. □

The proof of Lemma 6 is completed by combining Propositions 1, 2, 3 and 4. Then based on Lemma 6, we can show the main theorem of this section.

Theorem 1. *Given a sequence of at most L positive k-bit integers, where $L \leq U$ and $k = O(\lg \lg L)$. Then, there is a data structure of $kn + o(kn)$ bits of space, together with a precomputed table of $O(L^\epsilon)$ bits for any fixed $\epsilon > 0$, that supports* `sum` *or* `search` *in $O(\max\{\log_b n, 1\})$ time,* `update`, `insert` *or* `delete` *in $O(b)$ time, for any $\lg^{O(1)} L \leq b \leq L^{O(1)}$. The parameter n denotes the current size of the sequence.*

Proof. For ease of discussion, we assume at the moment that $k = o(\lg \lg L)$. We maintain two auxiliary data structures, one for `sum` and and one for `search`.

Assume that the sequence is $s_1, s_2, \ldots$. Firstly, we partition the elements into groups of size at least $(\lg L/\lg\lg L)^3$ and at most $2(\lg L/\lg\lg L)^3$, where elements in the same group are stored in consecutive locations. Each group is then maintained by Lemma 3.

Auxiliary data structure for `sum`: Let G_x denote the x-th group, and n_x denote its size. Then, to support `sum`, we construct a data structure $\mathcal{W}_1$ of Lemma 6 on all the n_x's. Notice that if a `search`(i) query on $\mathcal{W}_1$ returns x, it indicates that the element s_i is located in G_x. In order to support `sum`, we need to augment the WBB of $\mathcal{W}_1$ with the following things. Firstly, each leaf stores a pointer to the corresponding group. Secondly, in each node u, we store a TW-value (true-weight) which is the total sum of elements in those groups such that the corresponding leaves are under the left siblings of u.

To support `sum`(i) among the input sequence, we first perform a `search`(i) query on $\mathcal{W}_1$ to locate the group G_x that contains s_i. Then, the sum (say, y) of the TW-values on the traversal path indicates the total sum of elements in the groups preceeding G_x. Afterwards, a `sum`$(x-1)$ query on $\mathcal{W}_1$ returns a value i', which is the total number of elements preceeding G_x. To obtain the required answer, it remains to perform a `sum`$(i-i')$ query within G_x by Lemma 3, and add the result to y. The total time used is $O(\log_b n)$ by straightforward calculation.

For the space, $\mathcal{W}_1$ requires $o(n)$ bits (by setting $z = n/(\lg L/\lg\lg L)^3$ and $x = O((\lg L/\lg\lg L)^3)$ in Lemma 6), and it requires $o(n)$ bits for the augmented TW-values and the pointers in the leaves. For the groups, they occupy $o(n)$ bits in total. Thus, the auxiliary data structure for `sum` is of $o(n)$ bits.

Auxiliary data structure for `search`: For each group G, we partition them into units, so that the sum of each unit is at least $(\lg L/\lg\lg L)^3$ and at most $2(\lg L/\lg\lg L)^3$. Notice that each group has at most $2^k = O(\lg L)$ units. Each unit is maintained by Lemma 3.

Let U_x denote the x-th unit, and σ_x be the sum of its elements. Then, to support `search`, we construct a data structure $\mathcal{W}_2$ of Lemma 6 on all the σ_x's. Notice that if a `search`(j) query on $\mathcal{W}_2$ returns x, it indicates that the element $s_{\texttt{search}(j)}$ is located in U_x. In order to support `search`, we need to augment the WBB of $\mathcal{W}_2$ with the following things. Firstly, each leaf stores a pointer to the corresponding unit. Then, in each node u of $\mathcal{W}_2$, we store a TC-value (true-count) which is the total number of elements in those units such that the corresponding leaves are under the left siblings of u.

Now, to support `search`(j) among the input sequence, we first perform a `search`(j) query on $\mathcal{W}_2$ to locate the group U_x that contains $s_{\texttt{search}(j)}$. Then, the sum (say, y) of the TC-values on the traversal path indicates the total number of elements in the units preceeding U_x. Afterwards, a `sum`$(x-1)$ query on $\mathcal{W}_2$ returns a value j', which is the total sum of elements preceeding U_x. To obtain the required answer, it remains to perform a `search`$(j-j')$ query within U_x by Lemma 3, and add the result to y. The total time used is $O(\log_b n)$ by straightforward calculation.

For the space, $\mathcal{W}_2$ requires $o(n)$ bits (set $z = O(2^k n/(\lg L/\lg\lg L)^3)$ and $x = O(\lg L/\lg\lg L)^3$ in Lemma 6), while the augmented TC-values and the pointers in the leaves occupies $o(n)$ bits. For the units, they occupy $o(n)$ bits in total. Thus, the auxiliary data structure for `search` takes $o(n)$ bits.

Modification due to `update`, `insert` or `delete`: To modify the auxiliary structures due to `insert` or `delete`, we re-construct all the affected groups, and re-construct all the corresponding underlying units, and update $\mathcal{W}_1$ and $\mathcal{W}_2$ accordingly. Note that only constant number of groups are affected. Since construction time for the data structure of Lemma 3 depends on the size of the sequence stored, each group and all its underlying units can be re-constructed in a total of $O((\lg L/\lg\lg L)^2)$ time. For $\mathcal{W}_1$, it can be fixed in $O(b)$ time by Lemma 6. On the other hand, at most $O(\lg L/\lg\lg L)$ units are re-constructed, so that $\mathcal{W}_2$ can be modified in $O(b\lg L/\lg\lg L)$ time. By setting b to $b^{1/2}$, the total update time becomes $O(b)$ without affect the time for `sum` and `search`. Thus, the total time for both update operations is $O(b)$. Finally, notice that

`update` can be done by a `delete` followed by an `insert` (or vice versa), so it takes $O(b)$ time as well.
When $k = O(\lg \lg n)$: Notice that the above auxiliary data structures also work well for $k = O(\lg \lg L)$, except that the space required may become too large. This can be handled by a slight modification that, elements of A are partitioned into group of size at least $(\lg L/ \lg \lg L)^{O(1)}$ and at most $2(\lg L/ \lg \lg L)^{O(1)}$ instead. □

Corollary 1. *Given a sequence of positive k-bit integers, where $k = O(1)$. Then, there is a data structure of size $kn + o(kn)$ bits that supports* `sum` *and* `search` *in $O(\max\{\log_b n, 1\})$ time,* `update` *in $O(b)$ time, while* `insert` *and* `delete` *in amortized $O(b)$ time, for any $\lg^{O(1)} U \le b \le U^{O(1)}$. The parameter n denotes the current size of the sequence.*

Proof. Omitted (deferred to the full paper). □

4 Dynamic Bit Vector

This section investigates the tradeoff between query and update times for the dynamic bit vector problem. Our target aims at proving the following theorem:

Theorem 2. *Given a bit vector of length n. Together with an auxiliary data structure of $o(n)$ bits, we can support* `rank` *and* `select` *in $O(\log_b n)$ time, and* `update` *in $O(b)$ time, for any parameter b with $\lg n/ \lg \lg n \le b \le n$.*

By Lemma 2, we can show that the above theorem is correct for any parameter b that is between $\lg n/ \lg \lg n$ and $\lg^2 n$. Then, Lemma 1 implies that we have an auxiliary data structure of $o(n)$ bits that supports `rank` in $O(\log_b n)$ time and `flip` in $O(b)$ time, for $\lg^2 n \le b \le n$. To complete the proof of the above theorem, it remains to design an auxiliary data structure on the bit vector, that supports `select` in $O(\log_b n)$ time and `flip` in $O(b)$ time, for any $b \ge \lg^{O(1)} n$. The data structure is shown in following lemma.

Lemma 7. *Given a bit vector X of length n. Together with an auxiliary data structure of $o(n)$ bits of space, we can support* `select` *in $O(\log_b n)$ time and* `flip` *in $O(b)$ time, for any parameter b with $\lg^2 n \le b \le n$.*

Proof. We use a data structure to maintain all the positions p such that $X[p] = \mathtt{1}$. The positions are first partitioned into groups, where each group maintains its size to be at least $\lg^2 n$ and at most $2 \lg^2 n$. The $\Theta(\lg^2 n)$ positions within each group are organised as follows, so that for any j, we can report the j-th smallest position within the group in $O(1)$ time:

– We store sp and lp explicitly, where sp and lp denotes the smallest and the largest positions among the group.
– For the remaining positions,
  - if $lp - sp \ge \lg^4 n$, they are stored explicitly, so that for any j, the j-th smallest position can be returned immediately.
  - otherwise, we use Lemma 3 to store an auxiliary data structure of size $o(lp - sp)$ bits for the bit vector $X[sp..lp]$. Note that the j-th smallest position can be found by selecting the j-th `1` in $X[sp..lp]$.

Now, let G_i be the i-th group and s_i denote its size. Since each value of s_i is between $\lg^2 n$ and $2 \lg^2 n$, we can build a data structure $\mathcal{W}$ to maintain all the s_i's using Lemma 6.

To support `select`(j), we perform `search`(j) on $\mathcal{W}$. It returns x and indicates that G_x is the group that stores the j-th smallest position. Then, we obtain $j' = \mathtt{sum}(x-1)$ from $\mathcal{W}$, which represents the number of positions stored in the groups preceeding G_x. We can see that the required position of `select`(j) is stored as the $(j-j')$-th smallest value in G_x.

For time complexity, as there are $O(n/\lg^2 n)$ values stored in $\mathcal{W}$, `search` in this part takes $O(\log_b n)$ time. On the other hand, finding the $(j-j')$-th smallest position in G_x takes $O(1)$ time. Thus, the time for `select` follows.

As for the space, $\mathcal{W}$ occupies $O((n/\lg^2 n)\lg(b\lg^2 n)) = o(n)$ bits. For the groups, all the *sp* and *lp* values can be stored in $o(n)$ bits. Then, at most $O(n/\lg^4 n)$ of them would store the $\Theta(\lg^2 n)$ in-between positions explicitly. This occupies at most $o(n)$ bits. For the remaining groups, each requires $o(lp - sp)$ bits for the in-between positions. Again, these sum up to $o(n)$ bits. Thus, the space for the data structure follows.

Finally, to `flip` a bit in X, it corresponds to inserting or deleting a position in some group. Since each group contains $\Theta(\lg^2 n)$ positions, the affected group (or affected groups, in case we need to perform group splitting or merging to maintain the group size) can be updated by fixing at most $O(\lg^2 n)$ positions, or performing at most $O(\lg^2 n)$ `flip` in the auxiliary data structure of Lemma 3. Both can be done in $O(\lg^2 n)$ time. On the other hand, at most one s_i value is updated, inserted or deleted, so that $\mathcal{W}$ can be maintained in $O(b)$ time. Since $b \geq \lg^2 n$, the time for `flip` follows. This completes the proof of the lemma. □

By combining Lemmas 1, 2 and 7, Theorem 2 is proved.

5 Searchable Partial Sums when $k = O(\lg \lg n)$

This section shows that when $k = O(\lg \lg n)$, we can achieve the same tradeoff result as the *Dynamic Bit Vector* problem. Precisely, `sum` and `search` take $O(\log_b n)$ time, while `update` takes $O(b)$ time.

Notice that when $b < \lg^{O(1)} n$, the required tradeoff can be obtained by Lemma 2. In the remaining of this section, we thus focus on the case when $b \geq \lg^{O(1)} n$.

We first describe the data structure for `sum` among the elements of A.

Lemma 8. *Given a sequence of* n *non-negative* k*-bit integers that is stored in an array* A*, where* $k = O(\lg \lg n)$*. Together with an auxiliary data structure of size* $o(n)$ *bits, we can support* `sum` *in* $O(\log_b n)$ *time and* `update` *in* $O(b)$ *time, for any parameter* b *with* $\lg^2 n \leq b \leq n$*.*

Proof. (sketch.) We partition the elements of A into groups of size $\lg^2 n$, and store each group using Lemma 3. We build a complete b-ary tree, with each group represented by a leaf. Each node stores the LW-value, which is the sum of integers of A under all its left siblings.

Let G denote the group that contains $A[i]$. To support `sum`(i), we can first traverse from the root to the leaf that corresponds to G in the b-ary tree. Precisely, G is the $\lceil i/\lg^2 n \rceil$-th leaf in the tree. The sum (say, σ_1) of all the LW-values in the traversal path is thus equal to the total sum of the values in the groups preceeding G. Then, we perform a sum query within G, and obtain the sum (say, σ_2) of elements preceeding $A[i]$ within G. The required answer is thus equal to

$\sigma_1 + \sigma_2$, and the total time used is $O(\log_b n)$. For `update`, a straightforward calculation yields $O(b \log_b n)$ time, which can be reduced by setting b to $b^{1/2}$, without affecting other parts of the proof. Finally, the total space used is $o(n)$ bits. □

Next, we show how to support `search` and its update, as stated in the following lemma.

Lemma 9. *Given a sequence of n non-negative k-bit integers that is stored in an array A, where $k = O(\lg \lg n)$. Together with an auxiliary data structure of $o(n)$ bits of space, we can support* `search` *in $O(\log_b n)$ time and* `update` *in $O(b)$ time, for any parameter b with $\lg^{O(1)} n \le b \le n$.*

Proof. We partition the elements of A into groups of size $\lg^3 n$, and store each group using Lemma 3. Each group is given an index from 1 to $n/\lg^3 n$. We keep a dynamic bit vector $\mathcal{B}$ of length $n/\lg^3 n$ using Theorem 2 to mark all the *positive-sum* groups. Precisely, the i-th bit is marked `1` if the the sum of the elements in the index-i group is positive.

Let σ_t denote the sum of the elements in the t-th positive-sum group. Note that each value of σ_t is at most $2^k \lg^3 n = O(\lg^{O(1)} n)$, so that it can be stored in $O(\lg \lg n)$ bits of space. By setting $L = n$, we apply the data structure $\mathcal{C}$ of Theorem 1 on all the σ_t values.

Now, suppose that x is the element $A[\texttt{search}(j)]$. To support $\texttt{search}(j)$ among all the elements of A, we perform a $\texttt{search}(j)$ on $\mathcal{C}$, so that it returns which positive-sum group contains x. Suppose that it is the t-th positive-group. Then, we perform $\texttt{select}(t)$ on $\mathcal{B}$, which returns the index (say, p) of the group that contains x. Finally, a search within the index-p group returns the rank (say, r) of x within the group. Thus, the required answer is $p \lg^3 n + r$. As there are $O(1)$ steps, and each taking at most $O(\log_b n)$ time, the time bound follows.

The update time follows from straightforward calculation.

For the space, the structure $\mathcal{C}$ occupies $o(n)$ bits (since it maintains $n/\lg^3 n$ values, each occupies $O(\lg \lg n)$ bits), the bit vector $\mathcal{B}$ occupies $O(n/\lg^3 n)$ bits, and all the groups can be stored in $o(n)$ bits in addition to the array A. Thus, the total space of the auxiliary data structure is $o(n)$ bits. □

By combining Lemmas 2, 8 and 9, we have the following theorem:

Theorem 3. *Given a sequence of n non-negative k-bit integers, where $k = O(\lg \lg n)$. There is a data structure of $kn + o(kn)$ bits of space that supports* `sum` *and* `search` *in $O(\log_b n)$ time and* `update` *in $O(b)$ time, for any parameter b with $\lg n / \lg \lg n \le b \le n$.*

6 The General Searchable Partial Sums

In this section, we investigate the *Searchable Partial Sums* problem when $k = O(\lg U)$. We first show a lower bound on the time required for `search`, and then propose a data structure whose tradeoff achieves such a lower bound in the extreme case.

The lower bound is shown by a reduction from the static predecessor problem, which is defined as follows:

Definition 1. *The* Static Predecessor *problem maintains a set S of n integers from a universe $[1, U]$, and supports $\texttt{pred}(x)$ query which returns the largest integer in S that is at most x.*

Lemma 10. *In the worst case, any data structure of size* $O(kn)$ *bits for the* Searchable Partial Sums *requires* $\Omega\left(\sqrt{\lg n/\lg\lg n}\right)$ *time for* `search` *under the word RAM, irrespective of the* `update` *time.*

Proof. By reduction from the Static Predecessor problem. □

The above lemma implies that the general *Searchable Partial Sums* problem cannot achieve the same tradeoff result of $O(\log_b n)$ time for `sum` and `search` and $O(b)$ time for `update`, as in the case when $k = O(\lg\lg n)$. (Otherwise, when b is set to n, we get $O(1)$ time for `search`, which contradicts with Lemma 10.)

Finally, we claim the following result, whose proof is deferred to the full paper.

Theorem 4. *Given a sequence of* n *non-negative* k*-bit integers, where* $k = O(\lg U)$*. Then, there is a data structure of* $kn + o(kn)$ *bits of space that supports* `sum` *in* $O(\log_b n)$ *time and* `update` *in* $O(b)$ *time for any parameter* b *with* $\lg n/\lg\lg n \leq b \leq n$*. For* `search`*, the time depends on the range of* b *as follows:*

1. $O(\log_b n)$ *time when* $\lg n/\lg\lg n \leq b \leq \lg^{O(1)} n$,
2. $O(\lg n/\lg\lg n)$ *time when* $\lg^{O(1)} n \leq b \leq \lg^{\tau} n$,
3. $O(\tau \log_b n)$ *for* $\lg^{\tau} n \leq b \leq n$,

where τ *denotes the value of* $\min\left\{\lg\lg n \lg\lg U/\lg\lg\lg U, \sqrt{\lg n/\lg\lg n}\right\}$.

7 Conclusion

Raman, Raman and Rao [7] have raised the question that, for what value of k can we obtain a succinct data structure that supports `sum` and `search` in $O(\log_b n)$ time and `update` in $O(b)$ time.

In this paper, we are able to answer partly to the question by showing that such a structure is achievable when $k = O(\lg\lg n)$, while giving a negative example that such a structure does not exist in general. A remaining open question is: What is the bound of k for such structure to exist?

References

1. P. Beame and F. E. Fich. Optimal Bounds for the Predecessor Problem. In *Proc. ACM STOC*, pages 295–304, 1999.
2. P. F. Dietz. Optimal Algorithms for List Indexing and Subset Rank. In *Proceedings of Workshop on Algorithms and Data Structures*, pages 39–46, 1989.
3. M. L. Fredman and M. E. Saks. The Cell Probe Complexity of Dynamic Data Structures. In *Proc. ACM STOC*, pages 345–354, 1989.
4. R. Grossi, A. Gupta, and J. S. Vitter. High-Order Entropy-Compressed Text Indexes. In *Proc. ACM-SIAM SODA*, pages 841–850, 2003.
5. T. Hagerup. Sorting and Searching on the Word RAM. In *Proc. STACS*, pages 366–398, 1998.
6. G. Jacobson. Space-efficient Static Trees and Graphs. In *Proc. IEEE FOCS*, pages 549–554, 1989.
7. R. Raman, V. Raman, and S. S. Rao. Succinct Dynamic Data Structures. In *Proceedings of Workshop on Algorithms and Data Structures*, pages 426–437, 2001.
8. R. Raman, V. Raman, and S. S. Rao. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees and Multisets. In *Proc. ACM-SIAM SODA*, pages 233–242, 2002.
9. A. C. Yao. Should Tables Be Sorted? *Journal of the ACM*, 28(3):615–628, 1981.