Compressed Persistent Index for Efficient Rank/Select Queries

Wing-Kai Hon[1,⋆], Lap-Kei Lee[2,⋆⋆], Kunihiko Sadakane[3,⋆⋆⋆], and Konstantinos Tsakalidis[4]

[1] Department of Computer Science, National Tsing Hua University, Taiwan
[2] HKU-BGI Bioinformatics Algorithms & Core Technology Research Laboratory, University of Hong Kong, Hong Kong
[3] National Institute of Informatics, 2-1-2 Hitotsubashi, Tokyo 101-8430, Japan
[4] Department of Computer Science & Engineering, HKUST, Hong Kong

Abstract. We design compressed persistent indices that store a bit vector of size n and support a sequence of k bit-flip update operations, such that rank and select queries at any version can be supported efficiently. In particular, we present partially and fully persistent compressed indices for offline and online updates that support all operations in time polylogarithmic in n and k. This improves upon the space or time complexities of straightforward approaches, when $k = O(\frac{n}{\log n})$, which is common in biological applications. We also prove that any partially persistent index that occupies $O((n+k)\log(nk))$ bits requires $\omega(1)$ time to support the rank query at a given version.

1 Introduction

In this paper we consider the problem of maintaining persistently a compressed bit vector under (online and offline) bit-flip updates, such that *rank* and *select* queries (and even updates) can be supported at any version of the bit vector. We consider the word-RAM model of computation. Although many persistent implementations have been devised for specific data structures, such as deques, dictionaries, etc. [8], this is the first study of making a compressed data structure persistent. A potential application of our data structures can be found in *temporal* indexing of similar DNA sequences. Many existing index implementations are for a single DNA sequence and rely on rank/select queries over compressed bit vectors to support pattern searching queries, e.g., FM-index [5], wavelet tree [6]. By interpreting differences between sequences as offline updates and temporal modifications of the sequences as online updates, our structures provide the extra capability of temporal rank/select queries over any version of the sequences.

Specifically, let $B[1..n]$ be a bit vector of length n. For a bit $c \in \{0,1\}$ and an integer $i \in [1,n]$, the query operation $rank_c(B,i)$ returns the number of occurrences of c in the prefix $B[1..i]$ of B, and the query operation $select_c(B,i)$

⋆ W.K. Hon was supported by Taiwan NSC Grant 99-2221-E-007-123.
⋆⋆ L.K. Lee was supported by Hong Kong Research Grant Council HKU 713512E.
⋆⋆⋆ K. Sadakane was supported by JSPS KAKENHI 23240002.

F. Dehne, R. Solis-Oba, and J.-R. Sack (Eds.): WADS 2013, LNCS 8037, pp. 402–414, 2013.

returns the position of the i-th occurrence of c in B. In the dynamic case, the index also supports the *bit-flip*(B, i) update operation that flips the bit $B[i]$ from 1 to 0, or from 0 to 1. Yet such an index is *ephemeral*, meaning that an update operation creates a new version of B *without* maintaining previous versions.

In this paper, we are interested in maintaining a *persistent* index that moreover remembers all versions of B when updates are performed to it. In particular, we consider two notions of persistence: A *partially persistent* index allows only updates to the latest version of B and the other versions are read-only; the versions of B form a list called *version list*. A *fully persistent* index allows updates and queries to any version of B; the versions form a tree called *version tree*.

The ephemeral static and dynamic data structures proposed for this problem are all *succinct* (see [13,14] and references therein), namely their space usage is as close as possible to the information-theoretic lower bound. Following the literature, for a bit vector B of length n that stores m occurrences of 1-bits, this is n times the empirical zero-order entropy $H_0(B) = \frac{m}{n} \log \frac{n}{m} + \frac{n-m}{n} \log \frac{n}{n-m}$. However, a complication arises when independent update operations are maintained persistently, since after k updates at least $\log k + \log n$ bits are required in order to store respectively both the version number and the position of the bit flip of each update. Therefore, we define a persistent index to be *compressed* when it uses $nH_0(B_0) + o(n) + O(k \cdot \log(kn))$ bits of space, where B_0 is the initial version of the bit vector of length n. In other words, the initial bit vector is to be represented by a succinct data structure using space close to the information-theoretic lower bound, while we simultaneously maintain the information of each update using only $O(\log(kn)) = O(\log k + \log n)$ bits. Notice that after $k = \omega(\frac{n}{\log n})$ updates, the $O(k \log(kn))$ term dominates the space complexity and thus the structure occupies $\omega(n)$ bits. Then, we can straightforwardly modify a regular persistent binary tree [4] to support the operations in $O(\log n)$ time, using $O(n)$ *words*.

Therefore we focus on "small" sequences of $k = O(\frac{n}{\log n})$ updates wherein the structure occupies $O(n)$ *bits*. This is a typical scenario in biological applications: We want to store a set of related DNA strings together, so that pattern searching queries can be supported efficiently. We may think of one string as a modification of the other. Here, the number k of DNA mutations is much smaller than the length of a DNA string. For example, for a human genome k is in the order of millions, while its length is around 3 billion nucleotides [15]. We study the problem under two types of updates, namely offline and online updates. For *offline updates*, all the k updates (and thus all the k versions of B) are known in advance. For *online updates*, the updates to B arrive in an online fashion such that an update must be performed before the next update arrives.

Previous Results. In the word-RAM model, Raman et al. [13] present a static succinct data structure that supports rank and select queries in $O(1)$ time. Sadakane and Navarro [14] present the *range min-max tree*, a dynamic succinct data structure that supports all operations in $O(\frac{\log n}{\log \log n})$ time. If we utilize this structure and store every version explicitly, the space usage will degrade to $O(n)$ words after only $k = O(\log n)$ updates. On the other hand, if we maintain only the information relevant to an update operation and reproduce a queried

Table 1. Asymptotic time bounds for persistent rank, select and bit-flip operations, where n is the size of bit vector B, k is the number of updates/versions and ϵ is any positive constant. The fully persistent index for online updates occupies $nH_0(B_0) + o(n) + O(k \log n \log(kn))$ bits, while the other indices are compressed. † is amortized.

	Offline updates	Online updates
Partially persistent	$\frac{\log k}{\log\log k}$, $\log n(\frac{\log k}{\log\log k})$, –	$(\frac{\log k}{\log\log k})^2$, $\log n(\frac{\log k}{\log\log k})^2$, $\log^{4+\epsilon} k$
Fully persistent	$\frac{\log^2 k}{\log\log k}$, $\log n(\frac{\log^2 k}{\log\log k})$, –	$\log^3 n$, $\log^3 n$, $\log^2 n \log\log n^{\dagger}$

version by the sequence of updates that created it, then the query time has a linear dependence on k in the worst case.

There exist generic techniques to render a data structure persistent in the pointer machine [4], word-RAM [3,9] and external memory [1] models. It is natural to consider applying these techniques to the range min-max tree [14]. The node splitting technique of Driscoll et al. [4] is applicable to pointer-based structures of constant-size nodes, which is not the case for the range min-max tree. Alternatively, we can store the tree in arrays and make them persistent using techniques in [3,9]. However, the arrays are not succinct and the update time is only efficient in expectation.

Our Contributions. This paper presents partially and fully persistent compressed indices for bit vectors that support efficient rank and select queries under sequences of offline and online bit-flip updates (see Table 1). They improve the space usage of straightforward approaches, as long as the number of bit flips k is $O(\frac{n}{\log n})$, where n is the bit vector size. These are the first compressed persistent indices that support all operations in time polylogarithmic in n and k.

In Section 2 we present the partially persistent indices for offline and online updates. They are obtained by storing the initial bit vector in a static structure for rank and select queries [13], and maintaining the information relevant to every update operation in a static (respectively dynamic) structure that supports planar orthogonal range counting queries [7,11]. Then we show how to obtain the answers of rank and select queries to a particular version without reconstructing the queried version, but instead by interpreting them as range counting queries appropriately. We follow a similar approach in the case of the fully persistent index for offline updates (Section 3), where we moreover apply centroid path decomposition (see, e.g., [2]) to the version tree in order to efficiently determine the updates that have created a queried version. To obtain the fully persistent index for online updates (Section 4), we first present the *range sum tree*, a simplification of the range min-max tree [14] that is succinct and supports rank, select and bit-flip in $O(\log n)$ time. Then we parametrize the I/O-efficient technique for full persistence of [1] such that it can handle nodes of non-constant size in the word-RAM model, and we apply it to the range sum tree.

Finally, in Section 5 we prove a superconstant lower bound on the rank query time of any partially persistent index that supports offline bit-flip updates and

uses $O((n+k)\log(nk))$ bits of space. This is in contrast to the non-persistent setting, where there exist succinct representations of the bit vector that support rank queries in $O(1)$ time [13].

2 Compressed Partially Persistent Index

In this section, we present two compressed partially persistent indices for offline and online updates, respectively. Let k be the number of updates and let n be the size of the bit vector B. The main results are stated below.

Theorem 1. *There is a compressed partially persistent index for offline updates that occupies $nH_0(B_0)+o(n)+O(k\log(kn))$ bits, and supports at any version, rank queries in $O(\frac{\log k}{\log\log k})$ time and select queries in $O(\frac{\log n \log k}{\log\log k})$ time.*

Theorem 2. *There is a compressed partially persistent index for online updates that occupies $nH_0(B_0)+o(n)+O(k\log(kn))$ bits, and supports at any version, rank queries in $O((\frac{\log k}{\log\log k})^2)$ time, select queries in $O(\log n(\frac{\log k}{\log\log k})^2)$ time and accessing a bit in $O((\frac{\log k}{\log\log k})^2)$ time. An update at the latest version takes $O(\log^{4+\epsilon} k)$ time, for any constant $\epsilon>0$.*

2.1 Data Structure and Algorithm for Offline Updates

We now show Theorem 1. The compressed partially persistent index consists of two components. The first component is a succinct data structure for the initial bit vector B_0. We use the data structure of Raman et al. [13] that occupies $nH_0(B_0)+o(n)$ bits and supports rank and select queries on B_0 in $O(1)$ time.

Lemma 1. *[13] A bit vector $B_0[1..n]$ can be stored using $nH_0(B_0)+O(\frac{n\lg\lg n}{\lg n})$ bits to support in $O(1)$ time the queries $rank_c(B_0,i)$ and $select_c(B_0,i)$, for any $1\le i\le n$ and $c\in\{0,1\}$.*

The second component stores the information of the update for each version. We reduce the rank and select query to the problem of *planar range counting* which, given Z points on a $N\times N$ grid, asks for the number of points in a given range $[x_1,x_2]\times[y_1,y_2]$. We employ the data structure of JáJá et al. [7].

Lemma 2. *[7] Let Z points lie on an $N\times N$ grid. Planar range counting queries can be supported in $O(\frac{\log Z}{\log\log Z})$ time, using $O(Z\log N)$ bits.*

We define two grids G_0 and G_1 of size $\max(k,n)\times\max(k,n)$, such that an update on bit i of version B_{t-1} from 1 to 0, which creates the new version B_t, corresponds to the point (t,i) on grid G_0 (similarly, on G_1 for bit-flips from 0 to 1).[1] We maintain two data structures of Lemma 2 for the grids G_0 and G_1, respectively. They occupy in total $2\cdot O(k\log(\max(k,n))) = O(k\log(kn))$ bits. Thus, the total space of both components is the space stated in Theorem 1.

[1] For offline updates, we can determine if the bit is flipped from 0 or 1 at no cost.

Query Algorithm. We can answer the queries $rank_c(B_t, i)$ and $select_c(B_t, i)$, for any version t, position $1 \leq i \leq n$ and bit $c \in \{0, 1\}$, as follows. Let $count_c(t, i)$ be the number of points in the range $[0, t] \times [0, i]$ of G_c.

- $rank_c(B_t, i)$: First, we obtain $rank_c(B_0, i)$ from the succinct data structure for B_0. Then, we make two range counting queries on the data structures for grids G_c and G_{1-c} to obtain $count_c(t, i)$ and $count_{1-c}(t, i)$. Then $rank_c(B_t, i) = rank_c(B_0, i) + count_c(t, i) - count_{1-c}(t, i)$.
- $select_c(B_t, i)$: $select_c(B_t, i)$ is the smallest $j \in [1, n]$ such that $rank_c(B_t, j) = i$. We find such a j, by a binary search on $rank_c(B_t, j)$.

Lemma 3. *For any version t, bit position $1 \leq i \leq n$ and bit $c \in \{0, 1\}$, the above query algorithms correctly answer $rank_c(B_t, i)$ and $select_c(B_t, i)$ in $O(\frac{\log k}{\log \log k})$ time and $O(\frac{\log n \log k}{\log \log k})$ time, respectively.*

Proof. We prove the correctness of answering $rank_c(B_t, i)$ by induction on the version t. When $t=0$, since $count_c(0, i) = count_{1-c}(0, i) = 0$, we have $rank_c(B_t, i) = rank_c(B_0, i) + count_c(0, i) - count_{1-c}(0, i) = rank_c(B_0, i)$. Assume that for some version $t \geq 1$, $rank_c(B_{t-1}, i)$ can be correctly answered, i.e., $rank_c(B_{t-1}, i) = rank_c(B_0, i) + count_c(t-1, i) - count_{1-c}(t-1, i)$. Recall that for a partially persistent index, an update on B_{t-1} (i.e., version $t-1$ of B) is a single bit-flip on B_{t-1}, which creates the bit vector B_t. There are three cases: **(1)** If a bit in position $[i+1, n]$ is flipped, $rank_c$ does not change. Since $count_c$ and $count_{1-c}$ remain the same, $rank_c(B_t, i) = rank_c(B_{t-1}, i)$. **(2)** If a bit in position $[1, i]$ is flipped from $1-c$ to c, then $rank_c$ is increased by 1. We have the point (t, i) in grid G_c, so $count_c(t, i) = count_c(t-1, i)+1$, while $count_{1-c}$ is unchanged. Thus, $rank_c(B_t, i) = rank_c(B_{t-1}, i)+1$. **(3)** If a bit in position $[1, i]$ is flipped from c to $1-c$, then $rank_c$ is decreased by 1. The point (t, i) is in grid G_{1-c}, so $count_{1-c}(t, i) = count_{1-c}(t-1, i)+1$ while $count_c$ is unchanged. Thus, $rank_c(B_t, i) = rank_c(B_{t-1}, i)-1$. Therefore, $rank_c(B_t, i)$ is correctly answered for all version t. It takes $O(1)$ time to obtain $rank_c(B_0, i)$ and $O(\frac{\log k}{\log \log k})$ time to obtain both $count_c(t, i)$ and $count_{1-c}(t, i)$. The total time is $O(\frac{\log k}{\log \log k})$.

The correctness of $select_c(B_t, i)$ follows from its definition. The binary search makes at most $O(\log n)$ queries on $rank_c(B_t, j)$ for $j \in [1, n]$, and each takes $O(\frac{\log k}{\log \log k})$ time, which implies the stated time complexity. □

2.2 Data Structure and Algorithm for Online Updates

We now consider online updates and show Theorem 2. For online updates, we will define a new query $access(B_t, i)$ that returns bit i in B_t. Similarly to Section 2.1, we divide the compressed partially persistent index into two components. The first component is the succinct data structure for the initial bit vector B_0 given in Lemma 1. The second component stores the update for each version, utilizing a dynamic data structure for the planar range counting problem.

Specifically, for an online update on bit i of B_{t-1} that creates B_t, we need to add the point (t, i) to one of the grids G_0 and G_1 in an online fashion. Nekrich [11]

has presented data structures for the *dynamic* planar range counting problem, where points can be added to or removed from the grid dynamically.

Lemma 4. *[11] Let Z points lie on an $N \times N$ grid. Planar range counting queries can be supported in $O((\frac{\log Z}{\log \log Z})^2)$ time, and updates in $O(\log^{4+\epsilon} Z)$ time, for any constant $\epsilon > 0$, using $O(Z \log N)$ bits.*

In the case of online updates, the maximum version number, denoted by K, is not fixed. We can set K to some constant and double it, whenever the current version number k is equal to K. In this way, K is always at most $2k$, and thus a version number can be represented in $O(\log k)$ bits. Similarly to Section 2.1, we define two grids G_0 and G_1 of size $\max(K, n) \times \max(K, n)$, such that an update on bit i of B_{t-1} from 1 to 0 (that gives B_t) corresponds to the point (t, i) on grid G_0 (similarly, on G_1 for 0-to-1 bit-flips). Here, to determine if an update is a bit-flip from 0 to 1 or vice versa, we need to call $access(B_{t-1}, i)$.

We maintain two data structures of Lemma 4 for the grids G_0 and G_1, respectively, which occupy $2 \cdot O(k \log(\max(K, n))) = O(k \log(kn))$ bits of space in total. Thus, the total space of both components is the space stated in Theorem 2.

Query Algorithm. For any version t, position $1 \leq i \leq n$ and bit $c \in \{0, 1\}$, we answer $rank_c(B_t, i)$ and $select_c(B_t, i)$ in the same way as in Section 2.1. We answer $access(B_t, i)$, as follows. First, we obtain $access(B_0, i)$, which is the value of $B_0[i]$, from the succinct data structure for B_0. We make four planar range counting queries on grids G_0 and G_1 to obtain $count_0(t, i-1)$, $count_0(t, i)$, $count_1(t, i-1)$ and $count_1(t, i)$. Then we report $access(B_t, i)$ to be

$$access(B_0, i) + (count_1(t, i) - count_1(t, i-1)) - (count_0(t, i) - count_0(t, i-1)) \ .$$

The correctness of the rank and select queries follows directly from Lemma 3. Their time complexities are blown up by a factor of $O(\frac{\log k}{\log \log k})$, because we use the data structure of Lemma 4, instead of that of Lemma 2. Thus, the following lemma suffices to complete the proof of Theorem 2.

Lemma 5. *For any version t and bit position $1 \leq i \leq n$, the above query algorithm correctly answers $access(B_t, i)$ in $O((\frac{\log k}{\log \log k})^2)$ time. Furthermore, an update at the latest version takes $O(\log^{4+\epsilon} k)$ time, for any constant $\epsilon > 0$.*

Proof. Note that $count_1(t, i) - count_1(t, i-1)$ is the number of times bit i is flipped from 0 to 1 up to version t, while $count_0(t, i) - count_0(t, i-1)$ is the number of times bit i is flipped from 1 to 0 up to version t. Therefore, their difference is equal to the change of bit i from B_0 to B_t, and the correctness of $access(B_t, i)$ follows. The access query involves a call to $access(B_0, i)$ that takes $O(1)$ time, and four planar range counting queries that take $O((\frac{\log k}{\log \log k})^2)$ time, which implies the time complexity stated in Theorem 2.

For an online update on bit i of B_{t-1} that creates B_t, we need a query on $access(B_{t-1}, i)$ to determine which of grid G_0 or G_1 to add the point (t, i) to. This takes $O((\frac{\log k}{\log \log k})^2)$ time. By Lemma 4, adding the point (t, i) to a grid takes $O(\log^{4+\epsilon} k)$ time. Thus, each update takes $O(\log^{4+\epsilon} k)$ time in total. □

3 Compressed Fully Persistent Index for Offline Updates

This section considers offline updates and presents a compressed fully persistent index. Let k be the number of updates and let n be the size of the bit vector B.

Theorem 3. *There is a compressed fully persistent index for offline updates that occupies* $nH_0(B_0)+o(n)+O(k\log(kn))$ *bits, and supports at any version, rank queries in* $O(\frac{\log^2 k}{\log\log k})$ *time and select queries in* $O(\frac{\log n \log^2 k}{\log\log k})$ *time.*

The fully persistent index allows updates to any version. A version B_t is created by flipping a single bit in B_p for some $p<t$. Let T be the version tree.

Centroid Path Decomposition. We decompose the version tree T using centroid path decomposition (see, e.g., [2]), as follows. For any internal node u, let v be the child of u with the largest number of leaves in its subtree (ties are broken arbitrarily). We refer to edge uv as a *core edge*, and to non-core edges as *side edges*. A *centroid path* C is a maximal path connecting consecutive core edges. The root of C, denoted by $r(C)$, is the top-most node of C. We denote by $\Delta(T)$, the set of all centroid paths in T. The following property is well-known.

Property 1. Let T be a tree of k nodes with a centroid path decomposition. The path from the root of T to any node v traverses at most $\log k$ centroid paths.

Data Structure and Algorithm. The compressed fully persistent index consists of three components. The first component is the succinct data structure for the initial bit vector B_0 given in Lemma 1. The second component stores the version tree T and three pieces of auxiliary information for each node in T. In particular, for each version v, we maintain the version number $p(v)$ of its parent. We also assign a *node label* $\ell(v)$ from 1 to k to each node v in ascending order of their depth, such that the node labels are strictly increasing along the path from the root of T to any node v of T. Finally, every node v is in some centroid path C, and we define $f(v)$ to be the root $r(C)$ of C. We can store $p(v)$, $\ell(v)$ and $f(v)$ in three arrays of size k, which allows $O(1)$ time access, given the version number v. In total, the second component takes $O(k\log(kn))$ bits of space.

The third component stores the information of the update for each version, using the data structure for the planar range counting problem of Lemma 2, as follows. For each centroid path C, we define two grids $G_0(C)$ and $G_1(C)$ of size $\max(k,n)\times\max(k,n)$. Consider each update on a version p that creates a version $t>p$, where $t\in C$. If the update flips bit i of B_p from 1 to 0, there is a point $(\ell(t),i)$ on grid $G_0(C)$ (similarly, on $G_1(C)$ for 0-to-1 bit-flips). For each centroid path C, we maintain two structures of Lemma 2 for the grids $G_0(C)$ and $G_1(C)$, respectively. These data structures are associated with the node $r(C)$. For all centroid paths, this takes $2\cdot\sum_{C\in\Delta(T)} O(|C|\log(\max(k,n))) = 2\cdot O(k\log(\max(k,n))) = O(k\log(kn))$ bits of space. Thus, the total space of all components is the space stated in Theorem 3.

Query Algorithm. Consider any version t, position $1\le i\le n$ and bit $c\in\{0,1\}$. We answer the query on $select_c(B_t,i)$ by using $rank_c$ in the same way as in

Section 2.1. We now give the query algorithm for answering $rank_c(B_t, i)$. Let $count_c(C, t, i)$ be the number of points in the range $[0, t]\times[0, i]$ of $G_c(C)$.

- $rank_c(B_t, i)$: First, we obtain $rank_c(B_0, i)$ from the succinct data structure for B_0. Then, we consider all updates along the path from the root of T to version t. Let $U = (u_0{=}0, u_1, u_2, \ldots, u_{x-1}, u_x{=}t)$ be the path that contains x versions. Suppose U traverses y centroid paths in the order of $C_1, C_2, \ldots, C_y$. The roots of all these y centroid paths must be in U; we denote them by $u_{z(1)}, u_{z(2)}, \ldots, u_{z(y)}$. Note that $u_{z(1)} = u_0 = 0$.
  Let $count'_c = \sum_{j=1}^{y-1} count_c(C_j, \ell(u_{z(j+1)-1}), i) + count_c(C_y, \ell(t), i)$, and let $count'_{1-c} = \sum_{j=1}^{y-1} count_{1-c}(C_j, \ell(u_{z(j+1)-1}), i) + count_{1-c}(C_y, \ell(t), i)$. We compute them as follows. Since $t{=}u_x$ is in C_y, the root of C_y is $u_{z(y)}{=}f(t)$. Since $u_{z(y)-1}$ is in C_{y-1}, the root of C_{y-1} is $u_{z(y-1)}{=}f(u_{z(y)-1})$. We repeat the above to identify the roots of all the y centroid paths, and make $2y$ range counting queries on grids $G_c(C_j)$ and $G_{1-c}(C_j)$ for $1{\le}j{\le}y$, respectively, to compute the counts. Finally, $rank_c(B_t, i) = rank_c(B_0, i) + count'_c - count'_{1-c}$.

To establish Theorem 3, it suffices to prove the correctness and time complexity for the rank query, since for select they follow similarly to Lemma 3.

Lemma 6. *For any version t, position $1{\le}i{\le}n$ and $c{\in}\{0, 1\}$, the above query algorithm correctly answers $rank_c(B_t, i)$ in $O(\frac{\log^2 k}{\log\log k})$ time.*

Proof. It suffices to prove that counter $count'_c$ (resp. $count'_{1-c}$) counts correctly the updates along the path U that flip the bits in position $[1, i]$ from $1{-}c$ to c (resp. from c to $1{-}c$). Since each such flip contributes 1 (resp. -1) to $rank_c(B_t, i)$, $rank_c(B_t, i) = rank_c(B_0, i) + count'_c - count'_{1-c}$ will follow.

Recall that $count'_c = \sum_{j=1}^{y-1} count_c(C_j, \ell(u_{z(j+1)-1}), i) + count_c(C_y, \ell(t), i)$. For convenience, we set $z(y{+}1) = t{+}1$. We focus on the path $U_j = (u_{z(j)}, u_{z(j)+1}, \ldots, u_{z(j+1)-1})$ for some $1{\le}j{\le}y$. Then $U_j{\subseteq}C_j$. By the definition of node labels, we have that $\ell(u_{z(j)}){<}\ell(u_{z(j)+1}){<}\cdots{<}\ell(u_{z(j+1)-1})$ and all other nodes in C_j have a node label larger than $\ell(u_{z(j+1)-1})$. Thus, on the grid $G_c(C_j)$, $count_c(C_j, \ell(u_{z(j+1)-1}), i)$ correctly counts the updates along the path U_j that flip bits in position $[1, i]$ from $1{-}c$ to c. Summing over all j, it follows that $count'_c = \sum_{j=1}^{y} count_c(C_j, \ell(u_{z(j+1)-1}), i)$ correctly counts such bit flips made by the updates along the path U. The correctness of $count'_{1-c}$ follows similarly.

Regarding time complexity, it takes $y{\cdot}O(1) = O(y)$ time to identify the roots of the y centroid paths. By Lemma 2, it takes $2y{\cdot}O(\frac{\log k}{\log\log k}) = O(y{\cdot}\frac{\log k}{\log\log k})$ time for the $2y$ range counting queries. By Property 1, the number of centroid paths in U is $y{=}O(\log k)$. Therefore, the total time complexity is $O(\frac{\log^2 k}{\log\log k})$. □

4 Compressed Fully Persistent Index for Online Updates

This section considers online updates and presents a compressed fully persistent index. Let k be the number of updates and let n be the size of the bit vector B.

Theorem 4. *There is a compressed fully persistent index for online updates that occupies* $nH_0(B_0)+o(n)+O(k\log n\log(kn))$ *bits, and supports at any version, rank, select and access queries in* $O(\log^3 n)$ *worst case time. An update at any version takes* $O(\log^2 n\log\log n)$ *amortized time.*

Overview. To show Theorem 4, we first present the *range sum tree*, a simplification of the *range min-max tree* of Sadakane and Navarro [14], that supports rank, select and bit-flip in $O(\log n)$ time. Then we make it fully persistent using a generic method from [1], which is designed for the I/O model and can be applied to the word-RAM model with a modest blow-up on time.

Range Sum Tree. The range sum tree is a balanced binary tree T, where each node corresponds to a range $[i,j]$ of B and it stores i, j and a value $e(i,j)$ that represents the number of 1's in $B[i,j]$. We divide the bit vector B into segments of length $L=\log^2 n$ and each leaf of T corresponds to the range of a segment. Let $[i_z,j_z]$ be the range of a node z. An internal node z with left child u and right child v has the range $[i_u,j_v]$ and $e(i_z,j_z)=e(i_u,j_u)+e(i_v,j_v)$. Therefore, the number of nodes in T is $O(\frac{n}{\log^2 n})$ and each node needs $O(\log n)$ bits, which sums up to $O(\frac{n}{\log n})$ bits of space.

Each leaf node also stores the bits in $B[i,j]$ for the query, as follows. We further divide the length-L segment into $2\log n$ sub-segments of length $t=\frac{\log n}{2}$. A leaf node has $2\log n$ extra fields, each representing a sub-segment succintly [13]: Each sub-segment with x bits belongs to a class x of t-bitmaps. E.g., if $t=2$, class 0 is $\{00\}$, class 1 is $\{01,10\}$ and class 2 is $\{11\}$. As class x contains $\binom{t}{x}$ elements, we can use $\lceil\log\binom{t}{x}\rceil$ bits and $\lceil\log(t+1)\rceil$ bits respectively to represent its element index within the class and the class identifier. As shown in [13], all sub-segments take at most $nH_0(B)+O(\frac{n}{\log n})$ bits of space.

Let $P_{x,y}$ be the length-t sub-segment represented by element y of class x. We maintain three universal tables U_{rank}, $U_{select,0}$ and $U_{select,1}$ for each class, such that given class x, element index y and an integer $0\le i\le t$, $U_{rank}(P_{x,y},i)$ returns the number of 1's in $P_{x,y}[1,i]$; and $U_{select,0}(P_{x,y},i)$ (resp. $U_{select,1}(P_{x,y},i)$) returns the smallest index j such that $P_{x,y}[1,j]$ contains i 1's (resp. i 0's). These tables need $3\cdot O(2^t\cdot t\cdot\log t)=O(\sqrt{n}\log n\log\log n)=o(n)$ bits. Thus, the range sum tree takes $nH_0(B)+2\cdot O(\frac{n}{\log n})+o(n)=nH_0(B)+o(n)$ bits in total.

Query Algorithm. We traverse T to answer a query on any bit position $1\le i\le n$. Initially, we set z to be the root of T. Note that $[i_z,j_z]=[1,n]$. We traverse T depending on whether z is an internal node or leaf node as follows.

- *rank*$_1(B,i)$: We count the number of 1's in $[1,i]$ using a counter *count*$_1$ initiated to 0. **(1)** z is an internal node with left child u and right child v: If $i\in[i_v,j_v]$, we add $e(i_u,j_u)$ (i.e., the number of 1's in $[i_u,j_u]$) to *count*$_1$, and set $z=v$. If $i\in[i_u,j_u]$, we set $z=u$. Then we repeat this procedure. **(2)** z is a leaf node: Let $(S_1,S_2,\ldots,S_{2\log n})$ be the sub-segments of z. Suppose position i is in S_j. We make j queries to the universal tables U_{rank} to determine the number of 1's in $S_1,S_2,\ldots,S_{j-1}$ and S_j up to position i and add them to *count*$_1$. Finally, we return *count*$_1$ that is clearly the number of 1's in $[1,i]$.

– $select_1(B,i)$: We find the i-th 1-bit using a variable j (initiated to i) as follows. **(1)** z is an internal node with left child u and right child v: If $j{>}e(i_u,j_u)$, the i-th 1-bit is not in $[i_u,j_u]$. We decrease j by $e(i_u,j_u)$ and set $z{=}v$. If $j{\leq}e(i_u,j_u)$, the i-th 1-bit is in $[i_u,j_u]$ and we set $z{=}u$. Then we repeat this procedure. **(2)** z is a leaf node: If $j{>}e(i_z,j_z)$, the i-th 1-bit does not exist and we simply return $select_1(B,i){=}0$. Otherwise, let $(S_1,S_2,\ldots,S_{2\log n})$ be the sub-segments of z. We make at most $2\log n$ queries to the universal table $U_{rank}(S_\ell,t)$, where $t{=}\frac{\log n}{2}$, from $\ell{=}1,2,\ldots$, until we find an x, such that $\sum_{\ell=1}^{x} U_{rank}(S_\ell,t){\geq}j$.[2] Then the i-th 1-bit is in S_x. We make a query on $U_{select,1}(S_x, j{-}\sum_{\ell=1}^{x-1} U_{rank}(S_\ell,t))$ to obtain the position of B's i-th 1-bit in S_x. We return $select_1(B,i) = i_z{+}(x{-}1){\cdot}(\frac{\log n}{2}){-}1{+}U_{select,1}(S_x, j - \sum_{\ell=1}^{x-1} U_{rank}(S_\ell,t))$.

Note that the number of 0's in a range $[i,j]$ is equal to $(j{-}i{+}1){-}e(i,j)$. Thus, we can answer $rank_0(B,i)$ and $select_0(B,i)$ in a similar way, where for $select_0(B,i)$ we query $U_{select,0}$ instead of $U_{select,1}$. To answer $access(B,i)$, we traverse the path from the root to the leaf z containing $B[i]$, and identify the sub-segment S_x in z that contains $B[i]$, as described for $rank_1(B,i)$. Let $B[i]$ be bit j of S_x. We obtain $B[i] = U_{rank}(S_x,j){-}U_{rank}(S_x,j{-}1)$ with two queries on U_{rank}.

Regarding query time, each query takes $O(1)$ time for an internal node, and $O(\log n)$ time for a leaf node, since we make $O(\log n)$ queries on universal tables, where each takes $O(1)$ time. Since a path contains $O(\log(\frac{n}{\log^2 n}))=O(\log n)$ internal nodes and a leaf node, a query takes $O(\log n{\cdot}1{+}\log n) = O(\log n)$ time.

Updating Bit i**.** To update bit i, we first make a query on $access(B,i)$ to locate the leaf node z that contains $B[i]$. We update the sub-segment with $B[i]$ to a new sub-segment in $O(\log n)$ time, since the sub-segment is of length $t = \frac{\log n}{2}$. Then, we update each node u on the path from the root to z, as follows. If the update flips $B[i]$ from 0 to 1, we increase $e(i_u,j_u)$ by 1; otherwise, we decrease it by 1. The update time is $O(\log(\frac{n}{\log^2 n})) = O(\log n)$.

Lemma 7. *The range sum tree for a length-n bit vector B is a balanced search tree, where each internal node contains $O(1)$ fields and each leaf node contains $O(\log n)$ fields. It occupies $nH_0(B){+}o(n)$ bits and supports access, rank and select queries and updating a bit of B in $O(\log n)$ time by accessing $O(\log n)$ internal nodes and a leaf node (and for update, modifying a field in each of them).*

Fully Persistent Range Sum Tree. We apply on the range sum tree T the following result of [1] for the I/O model with disk block size of $\mathcal{B}$ words.

Lemma 8. *[1] Let T be a pointer-based ephemeral data structure that supports queries in $O(q)$ worst case I/Os and where updates make $O(u)$ modifications to T in the worst case. Given that every node of T occupies at most $O(1)$ blocks and has $O(1)$ maximum in-degree, T can be made fully persistent such that a query to a particular version is supported in $O(q)$ worst case I/Os, and an update to any*

[2] Such an x exists, because $j{\leq}e(i_z,j_z){=}\sum_{\ell=1}^{2\log n} U_{rank}(S_\ell,t)$.

version is supported in $O(u \log \mathcal{B})$ amortized I/Os. After performing a sequence of k updates, the fully persistent structure occupies $O(u \cdot \frac{k}{\mathcal{B}})$ blocks of space.

In the scheme of [1] for the above lemma, each I/O can be simulated by $O(\mathcal{B})$ RAM operations, so that the time complexity in the word-RAM model is $O(\mathcal{B})$ times that in the I/O model. We set the block size $\mathcal{B}=\log n$, such that a block contains $\log n$ words and each (internal or leaf) node of T occupies $O(1)$ blocks. Since all algorithms are implemented only by top-down traversals of T, the tree can be implemented such that each node has in-degree 1. We set $q=O(\log n)$, since accessing an internal node takes $O(1)$ time and a leaf node takes $O(\log n)$ time. By Lemma 7, the rank, select and access queries access $O(\log n)$ nodes and thus take $O(\log n \cdot q \cdot \mathcal{B})=O(\log^3 n)$ time. We set $u=O(\log n)$, since by Lemma 7, an update makes $O(\log n)$ modifications to T. Thus, the update time is $O(u \cdot \mathcal{B} \cdot \log \mathcal{B}) = O(\log^2 n \log\log n)$ amortized. The fully persistent structure occupies $O(\log n \cdot \frac{k}{\mathcal{B}})$ blocks $= O(\log n \cdot k)$ words $= O(\log n \cdot k \cdot \log(kn))$ bits, since the word size is $O(\log(kn))$. This gives Theorem 4.

5 Lower Bound

In this section, we show that even for offline updates, a partially persistent index for a length-n bit vector B that occupies $O((n+k)\log(kn))$ bits, where k is the number of updates, must answer the rank query at any version in $\omega(1)$ time.

Our proof is based on a reduction of the problem of *planar dominance counting*, which is defined as follows: on a grid $[1, N]\times[1, N]$ with N points, a dominance counting query (x, y) asks for the number of points in a given range $[0, x]\times[0, y]$. Pătraşcu [12] has shown that any static data structure of size $O(N)$ words must take $\Omega(\frac{\lg N}{\lg\lg N})$ time to answer a dominance counting query.

Theorem 5. *Let B be a length-n bit vector, where k offline bit-flip updates have been performed. A partially persistent index for B that occupies $O((n+k)\log(kn))$ bits of space must answer the rank query at any version in $\omega(1)$ time.*

Proof. Suppose for the sake of contradiction, the partially persistent index, denoted by I, can answer the rank query at any version in $O(1)$ time. We show how to use I in combination with y-fast tries [16] to answer a dominance counting query on a grid $G=[1, n]\times[1, n]$ with n points in $O(\log\log n)$ time, using only $O(n)$ words of space. This contradicts the lower bound of $\Omega(\frac{\lg n}{\lg\lg n})$ time for dominance counting queries [12] and thus proves the theorem.

Based on the n points on G, we construct a bit vector $B[1..n]$ and n offline bit-flip updates, as follows. All n bits in B are initially 0, which is the initial version B_0 of B. For each point (i, j) on G, suppose that among all the n points, i is the p-th smallest x-coordinate and j is the q-th smallest y-coordinate, where ties are broken arbitrarily. We construct an update operation that flips the p-bit of B_{q-1} from 0 to 1 to create version B_q.

We maintain the partially persistent index I for B that uses $O(2n\log n^2) = O(n\log n)$ bits, i.e., $O(n)$ words. In addition, we maintain a y-fast trie for all the

distinct x-coordinates X and another y-fast trie for all the distinct y-coordinates Y. These two y-fast tries occupy $O(|X|+|Y|)=O(2n)=O(n)$ words and allow us, given a dominance counting query (a, b), to determine in $O(\log \log n)$ time the predecessor $pred_X(a)$ of a in X (i.e., the largest element $c \in X$ such that $c \leq a$) and the predecessor $pred_Y(b)$ of b in Y. For each element $c \in X$ (resp. $c \in Y$), we also store the number $r_X(c)$ (resp. $r_Y(c)$) of points on G whose x- (resp. y-) coordinates are at most c. This requires $O(n)$ words.

To answer the dominance counting query (a, b), it is not hard to see that we can ask I for the rank of $r_X(pred_X(a))$ in version $r_Y(pred_Y(b))$ of B. The query time is $O(\log \log n)$ and the space is $O(n)$ words, completing the proof. □

6 Conclusion

In this paper we presented the first efficient compressed persistent indices for *bit* vectors that support temporal rank/select queries and *independent* bit-flip updates. Extending our results to handle general alphabets and/or correlated updates (that exhibit a smaller information-theoretic space lower bound) may find important applications in computational biology [10] and other fields. We leave as open the problem of designing a compressed fully persistent index for online updates. The rest of our structures can be improved by use of *succinct* (static or dynamic) data structures for planar range counting.

References

1. Brodal, G.S., Sioutas, S., Tsakalidis, K., Tsichlas, K.: Fully persistent B-trees. In: Proc. SODA, pp. 602–614 (2012)
2. Cole, R., Gottlieb, L.A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: Proc. STOC, pp. 91–100 (2004)
3. Dietz, P.F.: Fully Persistent arrays. In: Dehne, F., Santoro, N., Sack, J.-R. (eds.) WADS 1989. LNCS, vol. 382, pp. 67–74. Springer, Heidelberg (1989)
4. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. J. Comput. Syst. Sci. 38(1), 86–124 (1989)
5. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proc. FOCS, pp. 390–398 (2000)
6. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. SODA, pp. 841–850 (2003)
7. JáJá, J., Mortensen, C.W., Shi, Q.: Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 558–568. Springer, Heidelberg (2004)
8. Kaplan, H.: Persistent data structures. In: Handbook on Data Structures and Applications, ch. 31, pp. 31-1–31-26. CRC Press (2004)
9. Kopelowitz, T.: On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In: Proc. FOCS, pp. 283–292 (2012)
10. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. J. Comp. Biology 17(3), 281–308 (2010)
11. Nekrich, Y.: Orthogonal range searching in linear and almost-linear space. Comput. Geom. 42(4), 342–351 (2009)

12. Pătraşcu, M.: Lower bounds for 2-dimensional range counting. In: Proc. STOC, pp. 40–46 (2007)
13. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3(4), 43 (2007)
14. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Proc. SODA, pp. 134–149 (2010)
15. The 1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing. Nature 467(7319), 1061–1073 (2010)
16. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space $\Theta(n)$. Information Processing Letters 17(2), 81–84 (1983)