# Data Structures for SMEM-Finding in the PBWT

Paola Bonizzoni[1,*][0000−0001−7289−4988], Christina Boucher[2][0000−0001−9509−9725], Davide Cozzi[1][0000−0003−2439−0608], Travis Gagie[3][0000−0003−3689−327X], Dominik Köppl[4][0000−0002−8721−4444], and Massimiliano Rossi[2][0000−0002−3012−1394]

[1] University of Milano-Bicocca, Milano, Italy
paola.bonizzoni@unimib.it, d.cozzi@campus.unimib.it
[2] University of Florida, Gainesville, FL
{christinaboucher,rossi.m}@ufl.edu
[3] Dalhousie University, Halifax, NS, Canada
travis.gagie@dal.ca
[4] University of Muenster, Muenster, Germany
dominik.koeppl@uni-muenster.de
[*] Corresponding author: paola.bonizzoni@unimib.it

**Abstract.** The positional Burrows–Wheeler Transform (PBWT) was presented as a means to find set-maximal exact matches (SMEMs) in haplotype data via the computation of the divergence array. Although run-length encoding the PBWT has been previously considered, storing the divergence array along with the PBWT in a compressed manner has not been as rigorously studied. We define two queries that can be used in combination to compute SMEMs, allowing us to define smaller data structures that support one or both of these queries. We combine these data structures, enabling the PBWT and the divergence array to be stored in a manner that allows for finding SMEMs. We estimate and compare the memory usage of these data structures, leading to one data structure that is most memory efficient. Lastly, we implement this data structure and compare its performance to prior methods using various datasets taken from the 1000 Genomes Project data.

## 1 Introduction

The positional Burrows–Wheeler Transform (PBWT) was defined by Durbin [5] as a means for analyzing haplotype datasets. Hence, the input consists of $h$ sequences $S = \{S_1, \ldots, S_h\}$ containing $w$ biallelic sites corresponding to the same loci. The main idea is that specific loci are sequenced and it is determined if the position contains the major allele (denoted as 1) or has an alternate allele (denoted as 0). We will represent $S$ as a $h \times w$ binary matrix that is denoted as $\mathsf{M}$. The PBWT of $\mathsf{M}$ is another $h \times w$ binary matrix, denoted as $\mathsf{PBWT}[1..h][1..w]$, where the first column is the same as the first column of $\mathsf{M}$, and the $j$-th column of $\mathsf{PBWT}$ is obtained by stably sorting the rows of $\mathsf{M}[1..h][1..j-1]$ in colexicographic order (starting at column $j-1$) and then taking the final column of

the result. We can define this more formally by first defining the *$j$-th prefix array* ($\mathsf{PA}_j$), which is the co-lexicographic ordering of the prefixes $S_1[1..j], \ldots, S_h[1..j]$. It follows that an equivalent definition of the $\mathsf{PBWT}$ is $\mathsf{col}(\mathsf{PBWT})_1 = \mathsf{col}(\mathsf{M})_1$ and $\mathsf{col}(\mathsf{PBWT})_j[i] = \mathsf{col}(\mathsf{M})_j[\mathsf{PA}_{j-1}[i]]$ for all $i = 1..h$ and $j = 2..w$, where $\mathsf{col}(\mathsf{PBWT})_j$ denotes the $j$-th column of the $\mathsf{PBWT}$. As noted by Durbin, the $\mathsf{PBWT}$ is highly run-length compressible [5] when read in column-major order. One of the main motivations for the invention of the $\mathsf{PBWT}$ is that it enables set-maximal exact matches (SMEMs) to be found efficiently in haplotype data. Given the input sequences $S = \{S_1, \ldots, S_h\}$ and a pattern $P[1..w]$, we define $P[i..j]$, where $1 \leq i \leq j \leq w$, to be a SMEM if it occurs in one of the sequences in $S$ and one of the following holds: i) $i = 1$ and $j = w$; ii) $i = 1$ and $P[1..j+1]$ does not occur in $S$; iii) $j = w$ and $P[i-1..w]$ does not occur in $S$; iv) $P[i-1..j]$ and $P[i..j+1]$ does not occur in $S$. Given the $\mathsf{PBWT}$, a pattern $P[1..m]$, and a starting column $\ell$, the $\mathsf{PBWT}$ allows us to efficiently find all the sequences in $S$ that contain $P$ between columns $\ell$ and $\ell + m - 1$. If there are no such sequences then all the sequences that contain $P[1..m']$ between columns $\ell$ and $\ell + m' - 1$, where $P[1..m']$ is the longest prefix of $P$ that occurs in $S$, are returned. Durbin demonstrated that SMEMs can be identified in $\mathcal{O}(hw)$-time via the computation of the *divergence arrays*. Here, the *$j$-th divergence array* ($\mathsf{DA}$) stores, for each $i > 0$, the length of the longest common suffix between the $i$-th and $(i-1)$-st rows of $\mathsf{M}$ when the rows of $\mathsf{M}$ are sorted according to the co-lexicographic order of their prefixes up to the $j$-th column.

Although Durbin (and others, i.e., Li [10]) showed that run-length compressing $\mathsf{PBWT}$ leads to significant space improvement, there are only a few methods for storing the divergence array in a compressed manner. Cozzi et al. [3] provided one such approach that is based on casting the problem of finding SMEMs to computing matching statistics, and showing that computing matching statistics can be accomplished via building a data structure that mirrors that of Rossi et al. [13]. However, it is largely open how to store the $\mathsf{PBWT}$ alongside the auxiliary data structures needed to efficiently find SMEMs. In this paper, we generalize the approach of Cozzi et al. [3] by describing two queries (`start` and `extend`) that can be used in combination to find SMEMs in the $\mathsf{PBWT}$, and address the prevailing gap in the literature by providing a comprehensive list of smaller data structures that can be used to efficiently support `start` and/or `extend`. We show that these data structures can be combined in various ways to create data structures that store the $\mathsf{PBWT}$ in a manner that supports SMEM-finding.

We study the theoretical bounds of each data structure, and benchmark their memory consumption under a practical setting. This benchmarking leads to a solution that is deemed most performant. We fully implement this approach and compare it to the methods of Cozzi et al. [3] and Durbin [5] by building the data structure on increasingly larger haplotype datasets from the 1,000 genomes project data. We compare both the construction time and space, and the time and space to find SMEMs, allowing us to conclude about the practicality of the methods.

## 2 Preliminaries

**String definitions.** We assume that all input strings are binary strings since our application is biallelic haplotype data. Given a binary character $\mathsf{b}$, we denote the negation of $\mathsf{b}$ as $\neg\mathsf{b} = 1 - \mathsf{b}$. We denote the empty string as $\varepsilon$. We denote the length of $S$ by $|S|$. We denote the $i$-th prefix of $S$ as $S[1..i]$, the $i$-th suffix as $S[i..n]$, and the substring spanning position $i$ through $j$ as $S[i..j]$, with $S[i..j] = \varepsilon$ if $i > j$. Given two strings $S$ and $T$, we say that $S$ is lexicographically smaller than $T$ if either $S$ is a proper prefix of $T$ or there exists an index $i \geq 1$ such that $S[1..i] = T[1..i]$ and $S[i+1]$ occurs before $T[i+1]$. Lastly, for a string $S$, a binary character $\mathsf{c}$, and an integer $j$, we define $\mathsf{rank}$ query $S.\mathsf{rank}_{\mathsf{c}}(j)$ as the number of occurrences of $\mathsf{c}$ in $S[1..j]$, and the $\mathsf{select}$ query $S.\mathsf{select}_{\mathsf{c}}(j)$ as the position of the $j$-th $\mathsf{c}$ in $S$.

**RMQ, PSV, and NSV.** Given an array $\mathsf{A}[1..n]$ of integers, a range minimum query (RMQ) for two positions $i \leq j$ asks for the position $k$ of the minimum in $\mathsf{A}[i..j]$, i.e., $k = \text{argmin}_{k' \in [i..j]} A[k']$. We denote this query by $\text{RMQ}_{\mathsf{A}}(i, j)$. Given a position $i$ in $A$, we define the previous-smaller-value (PSV) as $\text{PSV}_{\mathsf{A}}(i) = \max(\{j : j < i, \mathsf{A}[j] < \mathsf{A}[i]\} \cup \{0\})$. We define the next-smaller-value (NSV) as $NSV_{\mathsf{A}}(i) = \min(\{j : j > i, \mathsf{A}[j] < \mathsf{A}[i]\} \cup \{n+1\})$.

**LCP and LCE.** Given two strings $S$ and $T$, we denote the length of the longest common prefix between $S$ and $T$ as $\mathsf{lcp}(S, T)$. Using this notation, we define the longest common prefix array of an input string $S$ of length $n$ (given its Suffix Array $\mathsf{SA}_S$) as LCP$[1..n]$ where LCP$[i] = \mathsf{lcp}(S[\mathsf{SA}_S[i]..n], S[\mathsf{SA}_S[i-1]..n])$ for all $i > 1$, and LCP$[1] = 0$. Given an input string $S$ of length $n$ and two integers $1 \leq i \leq n$ and $1 \leq j \leq n$, the Longest Common Extension (LCE) is the longest substring of $S$ that starts at both $i$ and $j$. Moreover, as we will discuss in this work, there are multiple data structures that efficiently support LCE queries for a string $S$.

**SLPs.** The concept of straight-line programs (SLPs) will be used in our work. An SLP is a representation of the input as a context-free grammar whose language is precisely the input string [11].

**Matching statistics in the PBWT.** Cozzi et al. showed that the problem of finding SMEMs in the $\mathsf{PBWT}$ can be cast into computing matching statistics for $P$, which is a generalization of Bannai et al. [1]. Given a pattern $P[1..w]$, the matching statistics of $P$ with respect to $S$ is an array $A[1..w]$ of $(\mathsf{row}, \mathsf{len})$ pairs such that for each position $1 \leq j \leq w$: (1) $S_{A[j].\mathsf{row}}[j - A[j].\mathsf{len} + 1..j] = P[j - A[j].\mathsf{len}+1..j]$, and (2) $P[j-A[i].\mathsf{len}..j]$ is not a suffix of $S_1[1..j], \ldots, S_h[1..j]$. Informally, for each position $j$ of the pattern $P$, $A[j].\mathsf{row}$ is one row of the input matrix $\mathsf{M}$ where a longest shared common suffix (of length $A[j].\mathsf{len}$) ending in position $j$ in the pattern $P$ and in $S_{A[i].\mathsf{row}}$ occurs.

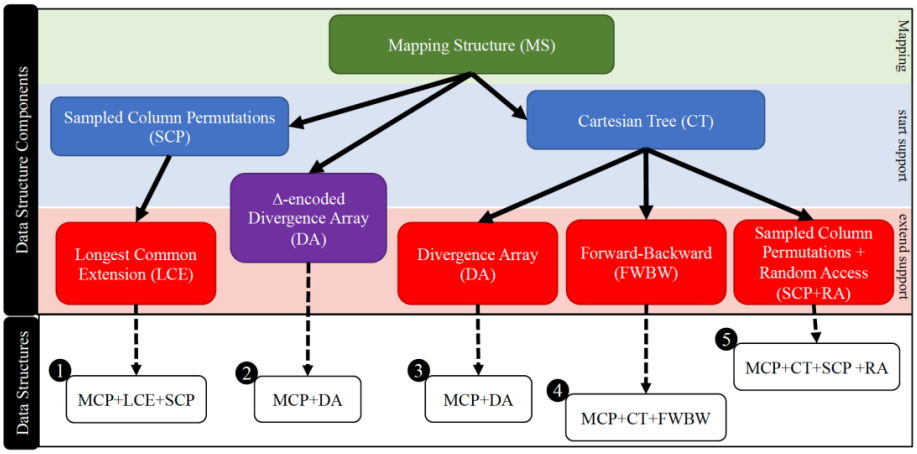


Fig. 1: An illustration of our components and data structures. The components are shown in colored boxes, and the data structures are shown in white boxes at the bottom.

## 3 Building Blocks for a PBWT Data Structure

In this section, we begin by defining two queries, called (`start` and `extend`), that are used to compute matching statistics in the PBWT. Then we define the smaller data structures, which we call components, that support `start` or `extend`—and in one case (i.e., the $\Delta$-encoded divergence array), can support both `start` and `extend`. These components are used to build data structures for SMEM finding in the PBWT. We show both the components and data structures in Figure 1. We note that we will frequently use $n = h \cdot w$ throughout this section to bound the time and space.

### 3.1 Queries Needed to Support SMEM-finding

We define two queries, referred to as `start` and `extend`, which can be used in combination to compute matching statistics, and hence, find SMEMs. If we let $i, j \in [1..w]$ be two integers such that $P[i..j]$ is a suffix of one of $S_1[1..j], \ldots, S_h[1..j]$ then the `extend` query returns that there exists the match of $P[i..j]$ to $P[i..j+1]$ if and only if $j < w$ and $P[i..j+1]$ is a suffix of one of $S_1[1..j+1], \ldots, S_h[1..j+1]$. The `start` query finds the smallest integer $i' \in [i..j]$ such that $P[i'..j]$ is a suffix of one of $S_1[1..j], \ldots, S_h[1..j]$. Hence, we compute the matching statistics as follows. We assume that we computed the matching statistics up to position $i \in [1..w]$, and use the `start` query to find the smallest $i' \in [i..w]$ such that $P[i'..i'+A[i].\mathsf{len}]$ is a suffix of one of $S_1[1..i' + A[i].\mathsf{len}], \ldots, S_h[1..i' + A[i].\mathsf{len}]$. By minimality of $i'$, we can set $A[j].\mathsf{len} = A[j-1].\mathsf{len} - 1$ for all $j \in [i+1..i'-1]$. Then we find the longest prefix $P[i'..k]$ that is also a suffix of one of $S_1[1..k], \ldots, S_h[1..k]$ using the `extend` query. We set $A[i'].\mathsf{len} = k - i' + 1$. Since $i' > i$, we can proceed by induction to compute the whole array of matching statistics.

### 3.2 Top Level: Mapping Structure

All the data structures that we present require a component data structure, which we call a mapping structure, that when given the position in $\mathsf{PBWT}$ of a bit $\mathsf{PBWT}[i][j]$ can return:

- the position in $\mathsf{col}(\mathsf{PBWT})_{j+1}$ of the bit immediately to the right of $\mathsf{PBWT}[i][j]$ in $\mathsf{M}$;
- the position in $\mathsf{col}(\mathsf{PBWT})_j$ of the last occurrence of $\neg\mathsf{PBWT}[i][j]$ above $\mathsf{PBWT}[i][j]$ (if it exists);
- the position in $\mathsf{col}(\mathsf{PBWT})_j$ of the first occurrence of $\neg\mathsf{PBWT}[i][j]$ below $\mathsf{PBWT}[i][j]$ (if it exists).

The first query corresponds to LF-mapping in the $\mathsf{BWT}$ and allows us to step from one column to the next one (to the right) in the $\mathsf{PBWT}$, staying in the same row in $\mathsf{M}$. The second and third queries correspond to how Rossi et al. [13] jump up or down, respectively, in the $\mathsf{BWT}$ when they find a mismatch. We implemented the mapping structure as a run-length compressed bitvector, occupying roughly $\mathcal{O}(r \log(n/r))$ bits and answering queries in $\mathcal{O}(\log \log n)$-time, where $r$ is the total number of runs in the columns of the $\mathsf{PBWT}$.

### 3.3 Second Level: Start Support

In this subsection, we provide a comprehensive discussion of all the data structures that support `start` queries.

**Sampled Column Permutations** If we use a Cartesian tree but neither the divergence array itself (encoded or unencoded) nor two instances of each component data structure, then it seems we need a way to find at least one occurrence of each SMEM in order to determine its length. We can use the sampled column permutations together with an analogue of Policriti and Prezza's [12] Toehold Lemma: for the bits at either end of each run in a column in the $\mathsf{PBWT}$, we store which rows in the input matrix they came from, using a total of roughly $2r \lg h$ bits; whenever we reach the right end of a SMEM and expand our search interval, the expanded interval must contain the first or last bit in some run of the bits we seek, and we learn from which row of the input matrix it came.

**Cartesian Trees.** If we store a representation of the shape of a Cartesian tree built upon the divergence array, then we can support RMQ, PSV and NSV queries on the divergence array. Yet, we note that these queries return only a position, and cannot easily support random access. We consider three representations of the tree shape: (1) an augmented balanced-parentheses (BP) representation occupying roughly $2n+o(n)$ bits [6] and answering queries in constant time; (2) a simple DAG-compressed representation (with each non-terminal storing the size of its expansion) answering queries in time bounded by its height; and (3) an interval-tree storing selected intervals corresponding to nodes in the Cartesian

tree and answering queries in constant time. We do not include Gawrychowski et al.'s compressed RMQ data structure [8] because we are not aware of an implementation and we see no easy way to estimate its space usage for the divergence array.

The constructions of the BP representation and DAG-compressed representations are standard, but our interval-tree structure needs some explanation. We query the Cartesian tree only while forward stepping through the PBWT and when our search interval contains only 0's and we want a 1, or when it contains only 1's and we want a 0. To proceed, we must ascend the tree and widen our interval (like ascending a prefix tree, discarding the early bits of our pattern) until it contains a copy of the bit we want. Notice our query interval corresponds to a node $v$ in the Cartesian tree, and the PBWT interval we seek corresponds to the lowest ancestor $u$ of $v$ whose interval is not unary. It follows that we need to store in our interval-tree only the PBWT intervals for nodes $u$ in the Cartesian tree such that the interval for at least one of $u$'s children is unary but $u$'s interval is not unary.

Since our intervals can nest but not otherwise overlap, we can store our interval-tree in a more space-efficient manner than usual: we write out a string with open-parens, close-parens and 0s, with each open-close pair indicating an interval and the number of 0s before, between and after them indicating its starting point, length and ending point; we encode that string as one bitvector with 0s indicating 0s and 1s indicating parens, and another bitvector with 0s indicating open-parens and 1s indicating close-parens (so the combination of the bitvectors is a wavelet tree for the string); and we store a BP representation of the tree structure of the stored intervals. If we store $k$ intervals, then our first bitvector has $n + 2k$ bits and $2k$ copies of 1, our second bitvector has $2k$ bits with $k$ copies of 0 and $k$ copies of 1, and the tree structure has $k$ nodes and so its BP representation takes $2k + o(k)$ bits. This means we use roughly $2k \lg \frac{n+2k}{2k} + 2k + 2k + o(k) = 2k \lg(n/k) + o(k \lg(n/k))$ bits, and can answer queries in constant time. We note that, since even our query intervals can nest but not contain or otherwise overlap any of our stored intervals, we can query with a single endpoint instead of a whole interval.

### 3.4 Third Level: Extend Support

In this subsection, we discuss all components that can support `extend` queries.

**Divergence Array.** The simplest possible data structure to support finding the length of each SMEM is to store the uncompressed divergence array, which was proposed by Durbin. The shortcoming of this is the large space requirements—as it would occupy space in bits roughly equal to the sum of the base-2 logarithms of all entries (with 2 added to each entry).

**Longest Common Extension** We consider the addition of an LCE data structure. Suppose we have arrived at column $j + 1$ and we know that the longest

suffix of pattern $P[1..j]$ that occurs in $\mathsf{M}$ ending at column $j$ has an occurrence immediately followed by $\mathsf{PBWT}[i][j+1]$, and we know which row of $\mathsf{M}$ that bit $\mathsf{PBWT}[i][j+1]$ comes from. If $P[j+1] = \mathsf{PBWT}[i][j+1]$ then the longest suffix of $P[1..j+1]$ that occurs in $\mathsf{M}$ ending at column $j+1$ has an occurrence ending with $\mathsf{PBWT}[i][j+1]$. Therefore, we assume $P[j+1] \neq \mathsf{PBWT}[i][j+1]$. By the definition of the $\mathsf{PBWT}$, there is an occurrence of the longest suffix of $P[1..j+1]$ that occurs in $\mathsf{M}$ ending at column $j+1$, ending either at the last occurrence of $P[j+1] = \neg\mathsf{PBWT}[i][j+1]$ above $\mathsf{PBWT}[i][j+1]$ (if it exists), or at the first occurrence of that bit below $\mathsf{PBWT}[i][j+1]$ (if it exists). We recall that the mapping structure allows us to quickly find these occurrences of that bit.

**Forward-Backward.** Suppose we use a Cartesian tree to maintain the invariant that our search interval in the $\mathsf{PBWT}$ contains all the bits immediately following occurrences of the longest suffix of the prefix of the pattern that we have processed so far, that occur in the desired columns of the $\mathsf{PBWT}$. If that search interval contains a copy of the next bit of the pattern, then we proceed by forward stepping, without consulting the Cartesisan trees. The only time we query the Cartesian trees is when the search interval does not contain a copy of the next bit of the pattern, meaning we have reached the right end of a SMEM. It follows that, using the mapping structure and the Cartesian trees, we can find the right endpoints of all the SMEMs. If we keep instances of all our components for the reversed input matrix, we can also find all the left endpoints of the SMEMs. Since SMEMs are maximal, they cannot nest, so we can easily pair up the endpoints and obtain the SMEMs. This doubles the time and space.

**Random Access.** Lastly, the simplest possible component is a compressed data structure of the original input that provides efficient random access to the input, which can obviously be used to find the length of a given SMEM. Although the total length of the SMEMs can be quadratic in the length of the pattern, the fact they cannot nest implies we need only a linear number of random accesses. In fact, if we combine a random access data structure with Cartesian trees then the number of random accesses is equal to the number of SMEMs, and the total length of the sequence that we extract from $\mathsf{M}$ is linear in the length of the SMEMs. There are many data structures that support random access to the input matrix $\mathsf{M}$, two notable ones are (a) an SLP of $\mathsf{M}$ (read row-wise) answering queries in $\mathcal{O}(\log n)$ time, and (b) a plain representation of $\mathsf{M}$ (with 8 bits packed into each byte) occupying roughly $n$ bits and allowing access to each bit in constant time.

### 3.5 $\Delta$-encoded Divergence Array

The last component we discuss is $\Delta$-encoded divergence array. As illustrated in Figure 1, we leave this component last since it can support both queries. To differentially encode ($\Delta$-encode) the divergence array, we store each entry of $\mathsf{DA}[i][j]$ with $i > 1$ as the difference $\mathsf{DA}[i][j] - \mathsf{DA}[i-1][j]$; for $i = 0$ we

always have $\mathsf{DA}[i][j] = 0$. If the $\mathsf{PBWT}$ is highly run-length compressible, read in column-major order, then the $\Delta$-encoded $\mathsf{DA}$ is small. To see why, consider that if $\mathsf{PBWT}[i..i+\ell-1][j]$ is a run of equal bits in the $j$-th column of the $\mathsf{PBWT}$ and $\mathsf{col}(\mathsf{PBWT})_{j+1}[i'..i'+\ell-1]$ are the bits immediately to their right in the input matrix, then $\mathsf{DA}[i'+k][j+1] = \mathsf{DA}[i+k][j]+1$ for $1 \leq k \leq \ell-1$. Therefore,

$$\mathsf{DA}[i'+k][j+1] - \mathsf{DA}[i'+k-1][j+1] = \mathsf{DA}[i+k][j] - \mathsf{DA}[i+k-1][j]$$

for $1 \leq k \leq \ell-1$, so the $\Delta$-encoded of $\mathsf{DA}[i'+1..i'+\ell-1][j+1]$ is the same as that of $\mathsf{DA}[i+1..i+\ell-1][j]$. It follows that, if there are $r$ runs in the columns of the $\mathsf{PBWT}$, then the (linearized) $\Delta$-encoded $\mathsf{DA}$ has a string attractor of size $\mathcal{O}(r)$ and, thus, it can be represented as a straight-line program occupying $\mathcal{O}(r \log^2 n)$ bits [9, Lemma 3.14].

Increasing the size of this SLP by a small constant factor, we can store at each non-terminal the length, sum, and minimum prefix sum of its expansion, and thus support random access, RMQ, PSV, and NSV queries on the divergence array in $\mathcal{O}(\log n)$-time. This is similar to how Gagie et al. [7, Lemma 6.2] used an SLP for their $\Delta$-encoded LCP array.

## 4 Composite Data Structures for the PBWT

We already described two data structures that efficiently support finding SMEMs in the previous section, namely, the "Mapping Structure + Cartesian Tree + Forward-backward" and "Mapping Structure + Cartesian Tree + Sample Column Permutations + Random Access". There are three other data structures that will be evaluated (Table 1), namely: (1) Mapping Structure + $\Delta$-Encoded Divergence Array; (2) Mapping Structure + Cartesian Tree + Divergence Array; and (3) Mapping Structure + LCE + Sampled Column permutations.

## 5 Experiments and Discussion

In this section, we provide experimental evaluations of our presented data structures. We begin by benchmarking the memory usage of our data structures. Based on these experiments, we fully implement one of these data structures and show the scalability of them on real data.

### 5.1 Comparison of Data Structures

*Experimental set-up.* We replicate the simulated dataset used by Durbin [5]. In particular, we run the Markovian coalescent simulator MaCS [2] with command line parameters `100000 2e7 -t 0.001 -r 0.001` to generate a haplotype matrix with 100,000 individuals and 360,000 sites. Next, we subsample the dataset with a parameter $\xi$ such that, given a column of length $h$ having $o$ ones, we skip this column if $o/h < \xi$. We set $\xi$ to be equal to 0.01, 0.03, 0.05, 0.08, and

| | Sample Parameter $\xi$ | | | | |
|---|---|---|---|---|---|
| Component | 0.01 | 0.03 | 0.05 | 0.08 | 0.10 |
| Mapping structure | 57M | 53M | 52M | 51M | 51M |
| $\Delta$-encoded divergence array | 479M | 452M | 435M | 426M | 418M |
| Cartesian tree | 472M | 472M | 458M | 420M | 402M |
| Longest common extension | 96M | 88M | 88M | 80M | 80M |
| Sampled column permutations | 80M | 76M | 76M | 76M | 76M |
| Divergence array | 125G | 92G | 77G | 64G | 58G |
| Random access | 96M | 88M | 88M | 80M | 80M |
| Data Structure | | | | | |
| `MAP + LCE + PERM` | **233M** | **217M** | **216M** | **207M** | **207M** |
| `MAP + DEDA` | 536M | 505M | 487M | 477M | 469M |
| `MAP + CT + FWBW` | 1.1G | 1.1G | 1.0G | 942M | 906M |
| `MAP + CT + DA` | 126G | 93G | 78G | 64G | 58G |
| `MAP + CT + PERM + RA` | 705M | 689M | 674M | 627M | 609M |

Table 1: The estimated size in bits of the combinations of component data structures that support SMEM-finding in the PBWT. M denotes megabytes and G denotes gigabytes. In boldface the best performance. We do not list Forward-Backward here as it is not a new data structure, only two instances of the Mapping Structure and the Cartesian Tree.

0.10, which results in the datasets having varying degrees of repetitiveness. See Table 1 for the size of the datasets. The haplotype matrix is publicly available at `http://dolomit.cs.tu-dortmund.de/tudocomp/pbwt_matrix.xz`. We ran all benchmarks on an Intel Core i3-9100 CPU (3.60GHz) with 128 GB RAM, running Debian 11.

*Implementation.* We implemented all methods in C/C++. The mapping structure was implemented using sparse bitvectors with a number of set bits equal to the number of runs. The differentially-encoded divergence array and the Cartesian tree were implemented with grammar compression. Forward and backward was implemented by building the data structures in both the forward and backward directions of the mapping structure. The sampled column permutations were obtained by sampling at run boundaries. The LCE data structure and the random access were implemented with as an SLP that answers LCE queries. All data structures are publicly available at `https://github.com/koeppl/pbwt`.

*Results.* We give the estimated sizes of the data structures that are compositions of these components in Table 1. We witness that `MAP+LCE+PERM` was the most performant, which was followed by `MAP+DEDA`. The performance of `MAP+DEDA` was somewhat surprising since it is similar to a structure suggested by Gagie et al. [7] that was but not implemented because it was thought to be impractical. We note that as $\xi$ increases the size of all the components and data structures decreases—this is intuitive since the datasets become less repetitive, resulting in fewer columns being selected. Hence, we see that the compression suffers for all the methods as $\xi$ increases but `MAP+LCE+PERM` maintains a lead.

### 5.2 Comparison of Methods on 1000 Genomes Project Data

*Experimental set-up.* We implement and evaluate `MAP+LCE+PERM` on the 1000 Genomes Project data by downloading the VCF files for the 1000 genomes project data and then converting these files to biallelic using `bcftools view -m2 -M2 -v snps` [4]. We consider increasingly larger datasets by selecting the panels for Chromosomes 22, 20, 18, 16 and 1 which have 5008 samples and a number of biallelic sites that range from ∼1 million to ∼6 millions. All datasets correspond to 4,908 individuals. All data are available at `https://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/`. We ran all experiments in this subsection on a machine with an Intel Xeon CPU E5-2640 v4 (2.40GHz) with 756 GB RAM and 768 GB of swap, running Ubuntu 20.04.4 LTS.

*Competing methods.* We compared against the PBWT implementation of Durbin [5], which are available at `https://github.com/richarddurbin/pbwt`. In detail, we ran both the `matchIndexed` and `matchDynamic` algorithms. We refer to these methods as `PBWT-index` and `PBWT-dynamic`, respectively. In addition, we compared against the methods of Cozzi et al. [3], which implements a mapping structure with sampled column permutations with the Thresholds data structure of Rossi et al. [13]. The method is referred to as $\mu$-PBWT. The computation of the matching statistics is analogous to Rossi et al. with one slight modification: the inverse of the mapping function is used to compute the lengths of the matching statistics. We refer to Cozzi et al. [3] for these details.

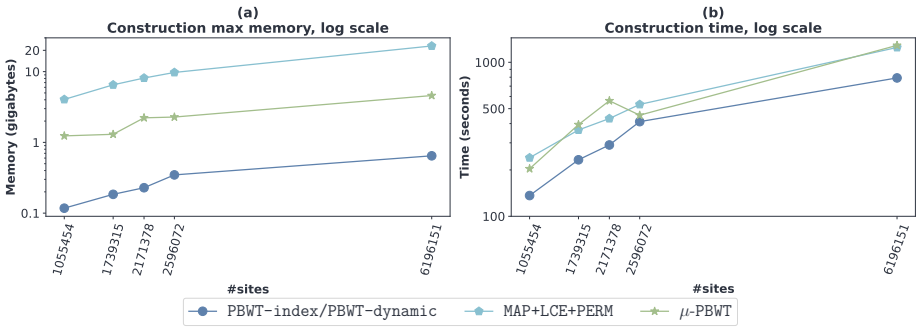


Fig. 2: Memory (a) and time (b) to construct the data structures underlying all methods for increasingly larger number of biallelic sites. Memory is reported in GB and time is reported in seconds.

*Results.* We give the maximum memory usage and time for constructing all the data structures in Figure 2. We note that the construction for `PBWT-index` and `PBWT-dynamic` is the same so it is reported once (as PBWT) in Figure 2, and that the constructed data structure is incomplete, meaning that additional indexes are needed for SMEM finding. This explains why the memory required for construction of the PBWT is small. We see that the method of $\mu$-PBWT requires

more memory than the PBWT but less memory than `MAP+LCE+PERM`. In terms of construction time, there was negligible difference between the performance of `MAP+LCE+PERM` and $\mu$-PBWT. PBWT had the most efficient construction time.

Next, we evaluated the performance of SMEMs-finding by first extracting 100 sequences from the input panels to be used query sequences. We illustrate the memory usage and the time required for SMEM-finding when all the query strings were given as input at once, which is shown in Figure 3 (a) and (b). Figure 3 (c) shows mean of the time required when each is given as an individual query, i.e., executing 100 queries one at a time. The peak memory usage to query all the sequences at once surpasses that of querying them individually. We obtained the following average number of SMEMs per 100 queries: 1,184 SMEMs for chromosome 22 (1,055,454 sites), 1,416 SMEMs for chromosome 20 (1,739,315 sites), 1,708 SMEMs for chromosome 18 (2,171,378 sites), 2,281 SMEMs chromosome 16 (2,596,072 sites) and 4,953 SMEMs for chromosome 1 (6,196,151 sites). `PBWT-dynamic` used the least memory when querying the whole set of queries but had opposing behavior doing one query at a time. It was fastest when the queries were given at once but slowest when the queries were given individually. Opposingly, `PBWT-indexed` required more memory than all other competing methods, requiring up to 20 times more memory. `PBWT-indexed` was the second slowest method when the queries were given at once but fastest when the methods were given individually. We see that `MAP+LCE+PERM` used less memory than `PBWT-MatchIndexed` and was at most 10 times slower than `PBWT-MatchIndexed` when the queries were given individually but was slightly slower than `PBWT-MatchIndexed` when the queries were given at all once. In addition, `MAP+LCE+PERM` was faster than `PBWT-MatchDynamic` when the queries were at once but slower than `PBWT-MatchDynamic` when queries were given individually. With respect to $\mu$-PBWT, `MAP+LCE+PERM` used slightly more memory and query time than $\mu$-PBWT. We note that `MAP+LCE+PERM` also has the advantage not requiring two passes on the query string, which makes it appropriate for online settings when the SMEMs can be found as the input is read in.

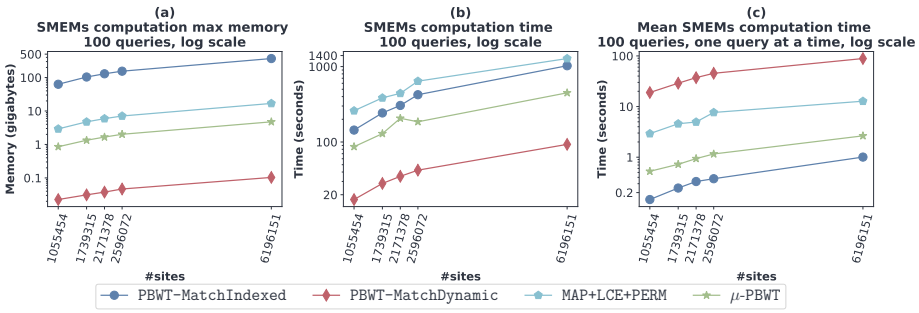


Fig. 3: Memory (a), time (b) and mean time for one query at a time (c) to compute SMEMs with 100 queries. In (c) the standard deviation values are very small so the corresponding error bars are omitted.

## 6 Conclusions

We presented and benchmarked a number of data structures that support SMEM-finding in the PBWT. Our experiments revealed that `MAP+LCE+PERM` was the most memory-efficient out of all data structures we presented. After fully implementing it, we showed that it is slightly slower and uses more memory than the method of Cozzi et al. [3]; however, we note that it has the advantage that it only requires one-pass over the query string, making it appropriate for the calculation of SMEMs in an online format.

**Acknowledgements** TR, MR, and CB were supported by NIH/NHGRI No. R01HG011392, and NSF IIBR No. 2029552. TR was supported NSERC No. RGPIN-07185-2020. CB was supported by NSF SCH No. 2013998. DK was supported by JSPS KAKENHI with No. JP21K17701 and JP23H04378. PB and DC were supported by Horizon 2020 with No. 872539.

## References

1. Bannai, H., Gagie, T., I, T.: Refining the r-index. Theoretical Computer Science **812**, 96–108 (2020)
2. Chen, G.K., Marjoram, P., Wall, J.D.: Fast and flexible simulation of DNA sequence data. Genome Research **19**(1), 136–142 (2009)
3. Cozzi, D., Rossi, M., Rubinacci, S., Köppl, D., Boucher, C., Bonizzoni, P.: $\mu$-PBWT: Enabling the storage and use of UK biobank data on a commodity laptop. bioRxiv pp. 2023–02 (2023)
4. Danecek, P., Bonfield, J.K., Liddle, J., Marshall, J., Ohan, V., Pollard, M.O., Whitwham, A., Keane, T., McCarthy, S.A., Davies, R.M., Li, H.: Twelve years of SAMtools and BCFtools. GigaScience **10**(2) (02 2021)
5. Durbin, R.: Efficient haplotype matching and storage using the positional Burrows–Wheeler transform (PBWT). Bioinformatics **30**(9), 1266–1272 (2014)
6. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM Journal on Computing **40**(2), 465–492 (2011)
7. Gagie, T., Navarro, G., Prezza, N.: Fully functional suffix trees and optimal text searching in BWT-runs bounded space. Journal of the ACM **67**(1), 2:1–2:54 (2020)
8. Gawrychowski, P., Jo, S., Mozes, S., Weimann, O.: Compressed range minimum queries. Theoretical Computer Science **812**, 39–48 (2020)
9. Kempa, D., Prezza, N.: At the roots of dictionary compression: string attractors. In: Proc. of ACM SIGACT Symposium on Theory of Computing (STOC). pp. 827–840 (2018)
10. Li, H.: BGT: efficient and flexible genotype query across many samples. Bioinformatics **32**(4), 590–592 (2016)
11. Lohrey, M.: Algorithmics on SLP-compressed strings: a survey. Groups - Complexity - Cryptology **4**(2), 241–299 (2012)
12. Policriti, A., Prezza, N.: LZ77 computation based on the run-length encoded BWT. Algorithmica **80**(7), 1986–2011 (2018)
13. Rossi, M., Oliva, M., Langmead, B., Gagie, T., Boucher, C.: Moni: A pangenomic index for finding maximal exact matches. Journal of Computational Biology **29**(2), 169–187 (2022)