IBB: Fast Burrows-Wheeler Transform Construction for Length-Diverse DNA Data

Enno Adler
Paderborn University, Germany

Stefan Böttcher
Paderborn University, Germany

Rita Hartel
Paderborn University, Germany

Cederic Alexander Steininger
Paderborn University, Germany

Abstract

The Burrows-Wheeler transform (BWT) is integral to the FM-index, which is used extensively in text compression, indexing, pattern search, and bioinformatic problems as *de novo* assembly and read alignment. Thus, efficient construction of the BWT in terms of time and memory usage is key to these applications. We present a novel external-memory algorithm called *Improved-Bucket Burrows-Wheeler transform* (IBB) for constructing the BWT of DNA datasets with highly diverse sequence lengths. IBB uses a right-aligned approach to efficiently handle sequences of varying lengths, a tree-based data structure to manage relative insert positions and ranks, and fine buckets to reduce the necessary amount of input and output to external memory. Our experiments demonstrate that IBB is 10% to 40% faster than the best existing state-of-the-art BWT construction algorithms on most datasets while maintaining competitive memory consumption.

2012 ACM Subject Classification Information systems → Data compression

Keywords and phrases burrows-wheeler transform, self-indexes, external-memory

Digital Object Identifier 10.4230/LIPIcs.SEA.2025.2

Related Version *Full Version*: `https://arxiv.org/abs/2502.01327`

Supplementary Material *Software*: `https://github.com/adlerenno/ibb` [1]
archived at `swh:1:dir:42d4aa3c09d658e12540f4090749c8c3d7966642`

1 Introduction

The Burrows-Wheeler transform (BWT) [5] is a widely used reversible string transformation with applications in text compression, indexing, and pattern search [10]. The BWT reduces the number of equal-symbol runs for data compressed with run-length encoding and allows pattern search in time proportional to the pattern length [10]. Because of these advantages and the property that the BWT of a string S can be constructed and reverted in $\mathcal{O}(|S|)$ time and space, the BWT plays an important role in computational biology, for example, in *de novo* assembly [24, 12] and short-read alignment. The tools BWA [15], Bowtie2 [13, 12], MICA [19], and SOAP2 [16] showcase the importance of the BWT.

Various BWT construction algorithms were created to fit the diverse properties of strings or DNA sequences. For example, BCR [3] was designed for short-read BWT construction, whereas ropeBWT2 [14] focuses on long reads. See Puglisi et al. [23] and Dhaliwal et al. [6] for an overview of suffix sorting algorithms; in Section 2 we summarize the state-of-the-art of BWT construction for DNA sequences.

In this paper, we present our new BWT construction algorithm, *Improved-Bucket Burrows-Wheeler transform* (IBB), which is especially designed for collections of strings W_i of highly diverse length. In Table 1, some of the listed datasets have very diverse length distributions.

Herein, our main contributions are as follows:

- An algorithm, IBB, to construct the BWT for DNA sequences of datasets with high diversity in length.[1]
- An extensive comparison of state-of-the-art BWT construction algorithms regarding the construction time and memory usage of BWT construction.[2] We show that IBB using a RAM disk is 10% to 40% faster than all other algorithms on most datasets while also using an amount of RAM smaller than most other approaches.

2 Related Work

The BWT [5] is key to many applications in bioinformatics such as short read alignment and *de novo* assembly. Thus, there is a broad field of ideas how to compute the BWT for a collection of DNA strings. As the BWT can be computed by taking the characters from the suffix array at the position before the suffix, suffix array construction algorithms (SACAs) like divsufsort [11], SA-IS by Nong et al. [21], gSACA-K by Louza et al. [17], or gsufsort by Louza et al. [18] can compute the BWT. Many SACAs rely on induced suffix sorting, which iteratively induces the order of suffixes from a sorted suffix subsets. Induced suffix sorting achieves its popularity by being a practically efficient linear-time suffix sorting algorithm.

The grlBWT method by Díaz-Domínguez et al. [7] is an example for a BWT construction algorithm that uses induced suffix sorting. It also uses run-length encoding and grammar compression for intermediate results, which results in a faster BWT construction.

The approach eGap by Egidi et al. [8] uses a divide and conquer strategy on the input collection: It divides the collection into subcollections and computes the BWT for the subcollection using gSACA-K [17]. After creating the BWT for each subcollection, eGap merges them to get the final BWT.

BigBWT by Boucher et al. [4] and r-pfbwt by Oliva et al. [22] use prefix-free parsing to reduce the size of the input. Prefix-free parsing represents the input by a parse and a dictionary, which is a prefix-free set. In prefix-free sets, no two words from the set are prefixes of each other. Therefore, the lexicographical order of words can be determined without relying on the word length and thus, the order of two suffixes starting with words from the prefix-free set is determined by these words. BigBWT uses prefix-free parsing on the input once. The approach r-pfbwt extends BigBWT by applying prefix-free parsing on the parse again, which recursively reduces the necessary space to represent the input.

All prior mentioned approaches use a concatenation of the input and a single-string construction in different ways. In contrast, using the *Last-to-First-Mapping* (LF-Mapping), which is normally used to inverse the BWT, poses different approaches as well: The BCR algorithm by Bauer et al. [3] inserts one character of each sequence per iteration for a collection of short DNA reads. The LF-mapping together with current insert position and the partially constructed part of the BWT is used to compute the next insert position of the next symbol of a sequence. BCR partitions these partially constructed BWTs into buckets, where one bucket contains all symbols prior to suffixes starting with the same symbol. BCR is an external algorithm, as it stores the buckets in external memory. Ropebwt and ropebwt2

[1] The implementation is available at `https://github.com/adlerenno/ibb`.
[2] The test is available at `https://github.com/adlerenno/ibb-test`.

by Li [14] resemble BCR, but they employ $B+$ trees for the buckets and are in-memory algorithms. Our approach, IBB, uses the LF-mapping as well, but introduces finer buckets and uses right-alignment on the input collection to reduce the size of partially constructed BWTs. IBB uses external memory to store the buckets. To have a predefined number of used files for representation of the buckets, we use static binary trees instead of the growing $B+$ trees of ropebwt2.

3 Preliminaries

In Appendix A, there is a list of symbols used in this paper.

We define a string S of length $|S| = n$ over $\Sigma = \{A, C, G, T\}$ by $S = S[0] \cdot S[1] \cdots S[n-1]$ with $S[i] \in \Sigma$ for $i < n$. We write $S[i, j] = S[i] \cdot S[i+1] \cdots S[j]$ for a substring of S, $S[i, j] = \epsilon$ if $i > j$, and $S[i..] = S[i, |S| - 1]$ for the suffix starting at position i. We define $rank_S(x, c) = |\{i < x : S[i] = c\}|$ as the number of times c occurs in $S[0, x-1]$. Next, we define $count_S(c) = |\{0 \leq i < |S| : S[i] < c\}|$ to be the number of characters in S that are lexicographically smaller than c.

Given a collection $(S_i)_{0 \leq i < m}$ of m strings of any length over Σ, let $M = max(\{|S_i| : 0 \leq i < m\})$, and we set $W = S_0\$_0 \dots S_{m-1}\$_{m-1}$ with $\$_0 < \cdots < \$_{m-1} < A < C < G < T$. We call the elements S_i of $(S_i)_{0 \leq i < m}$ words.

The Burrows–Wheeler transform $BWT(W)$ [5] of W can be computed by taking the last column of the sorted rotations of W. We define the LF-Mapping as

$$LF(i) = rank_{BWT(W)}(i, BWT(W)[i]) + count_{BWT(W)}(BWT(W)[i]).$$

For $0 \leq j < m$, the string S_j is obtainable from $BWT(W)$ by computing

$$BWT(W)[LF^{|S_j|-1}(j)] \dots BWT(W)[LF(j)] \cdot BWT(W)[j] = S_j$$

right to left and by aborting at $BWT(W)\left[LF^{|S_j|}(j)\right] = \$_{j-1}$ or $BWT(W)\left[LF^{|S_0|}(0)\right] = \$_{m-1}$ in case of $j = 0$ because the length $|S_j|$ is unknown. Since IBB does not need to differentiate the $\$_j$ symbols, we can simply write \$ for each of them.

4 IBB

First, we give a brief, high-level overview of our algorithm IBB before we describe technical details in the subsections. IBB inserts the characters of each word in iterations, such that in each iteration up to one character of every word is inserted. Because we have words of different length, we start a word as late as possible; consequently, all words insert their last character in the last iteration.

Each iteration consists of four steps: Updating the buckets, traversing the trees, inserting into the leaves and sorting the words. During the first three steps, we have two tasks: First, we convert the current insert position into an insert position for the leaf and second, we compute the insert position for the next iteration. The insert position into the partial BWT can be computed using the LF-Mapping, which consists of a rank expression and a count expression. In Subsection 4.3, we show that we can omit computing and adding the count if we use buckets. In Subsection 4.4, we introduce four binary trees to efficiently compute the rank over the partial BWT. Both, the buckets and the trees, divide the partial BWT into shorter fragments, which we save individually on external memory. The last step of each iteration is to sort the words according to their next insert position.

4.1 Right Alignment

We construct $BWT(W)$ without concatenating W and without using a single-string BWT construction algorithm. Instead, like BCR and ropeBWT2, we use iterations $t = 0, \ldots, M$. In each iteration, we insert at most one character of each word to compute a partial BWT string $bwt(t)$. Furthermore, like BCR and ropebwt2, we insert the characters of each word in reverse order. However, we do not start to insert the last character of each word S_j in iteration $t = 0$. Instead, we begin inserting characters of longer words in earlier iterations with the goal to terminate the insertion after the insertion of the first symbol of each word S_j in the second last iteration $t = M - 1$ followed by the insertion of a \$ character for each word S_j in the last iteration $t = M$.

As we will process the strings S_j from the end to the beginning, in each iteration t, we access the symbols $S_j[M - 1 - t]$ of strings S_j with $|S_j| + t > M$. In the following, we write the shortcut $W_j[t]$ for $S_j[M - 1 - t]$. We also set $W_j[M] =$ \$ for all words W_j. Figure 1a shows the words $W_0, \ldots, W_8$ already in reverse, right-aligned order during the iterations $t = 0, \ldots, 10$.

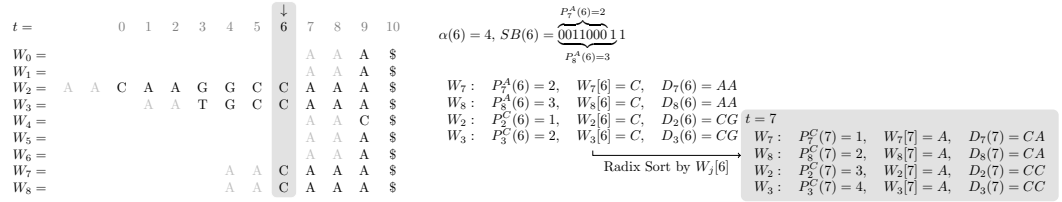
Our insertion order has two advantages: First, we keep the partial $bwt(t)$ smaller compared to starting all words in iteration $t = 0$, which allows faster computation of $bwt(t + 1)$ from $bwt(t)$. Second, we only have to consider the 4 characters A, C, G, and T instead of the 5 characters A, C, G, T, and \$ in $bwt(t)$ for all $t = 0, \ldots, M - 1$, which allows us to use smaller data structures.

The words $W_0, \ldots, W_m$ are not necessarily ordered by their length and swapping their order would change $BWT(W)$. We start inserting a word W_j in iteration $t = M - |W_j|$ and call the word *active* then. Let the number of currently active words in iteration t be $\alpha(t)$. When we start inserting a word W_j, we need to compute the insert position $P_j(M - |W_j|)$ for W_j in iteration $t = M - |W_j|$, because the insertion of shorter words with an index $i < j$ in $(W_i)_{0 \leq i < m}$ has not yet started. Our solution to this insert-position-problem uses a bit-vector $SB(t)$ of length m. Initially, $SB(0)[j] = 0$ for all $0 \leq j < m$ and we set $SB(t)[j] = 1$ when W_j starts in iteration $t = M - |W_j|$. After updating $SB(t)$ in an iteration $t = M - |W_j|$, the insert position $P_j(t)$ of $W_j[t]$ is $rank_{SB(t)}(j, 1)$, which counts the number of active words W_z with index $z \in [0, j - 1]$. For example, in iteration $t = 6$, the word W_7 is started. Its insert position shown in Figure 1b is $P_7(6) = rank_{SB(6)}(7, 1) = 2$.

4.2 Insert Position

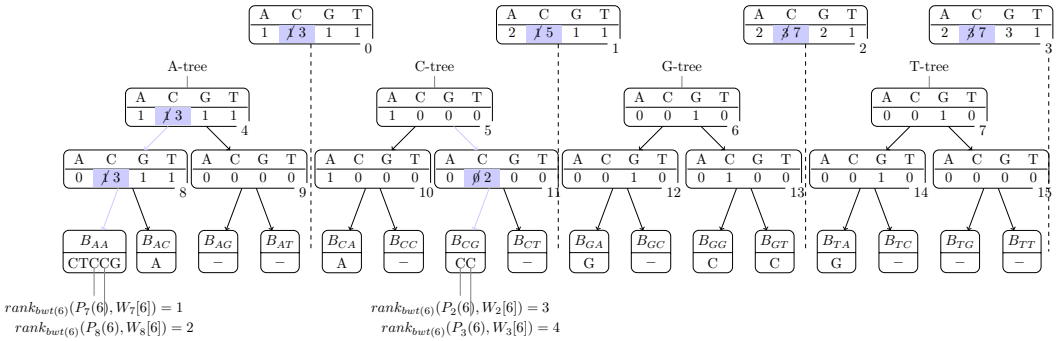
As in Ferragina et al. [9], the relative order of the symbols in $bwt(t)$, $t \geq 0$, is unchanged when inserting the new symbols to get $bwt(t + 1)$. Thus, given the last insert position $P_j(t)$ of character $W_j[t]$, $P_j(t + 1)$ can be obtained from an adjusted LF-Mapping. The adjustment regards the number of active words $\alpha(t+1)$ in iteration $t+1$ and is necessary, because $bwt(t)$, $t < M$, is not a valid BWT due to the missing \$ symbols. We need to adjust by $\alpha(t + 1)$ instead of $\alpha(t)$ because within a non-partial BWT, every active word in iteration $t + 1$ would need a \$ symbol.

The next insert position is $P_j(t + 1) = LF(P_j(t)) + \alpha(t + 1)$ for W_j in iteration $t + 1$ for $t < M$. In the last iteration $t = M$, we do not calculate any next insert positions, because we terminate after iteration $t = M$. Inserting the next character of each active word W_j into $bwt(t)$ – sorted according to the insert positions $P_j(t)$ – yields $bwt(t + 1)$.

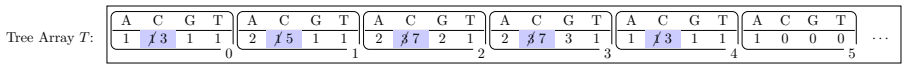


(a) The Words W_j are right aligned. The grey As in front of a word W_j do not belong to W_j; the As are the predecessor sequence, which are introduced by AW_j.

(b) Active words of iteration $t = 6$ and $t = 7$, their insert positions P, their characters to insert, and their predecessor sequences D. The bitvector SB is used to obtain the insert positions for the new started words W_7 and W_8.



(c) Tree structure for $k = 2$. Blue cells indicate an increase of the canceled value in iteration 6.



(d) Tree array T. The numbers in the lower right corners of boxes are the indices within the T array.

Figure 1 IBB structure for $k = 2$. All subfigures show the iteration $t = 6$ of the construction of the words in (a).

In the example of Figure 1c, the next insert position $P_3(7)$ of W_3 in iteration $t + 1 = 7$ given $P_3(6) = 8$ and $bwt(6) = CTCCGAACCGCCG$ is

$$\begin{aligned} P_3(7) &= LF(P_3(6)) + \alpha(7) \\ &= rank_{CTCCGAAC\ldots}(8, C) + count_{CTCCGAACCGCCG}(C) + \alpha(7) \\ &= 4 + 2 + 4 = 10. \end{aligned}$$

4.3 Buckets

BCR [3] and ropeBWT2 [14] segment $BWT(W)$ into buckets B_c based on the number of a character c in W. Formally, $B_c[i] = BWT(W)[i + count_W(c)]$ and $BWT(W) = B_\$B_AB_CB_GB_T(B_N)$ in case of ropeBWT2s alphabet that includes N additionally. We call the buckets B_c *level-1 buckets* and refine these buckets by looking at the first k characters instead of one character c. Hereby, k with $2 < k < log_4(|W|)$ is a fixed natural number. As shown in Figure 1c for $k = 2$, we have the *level-k buckets* $B_{c_1\cdots c_k}$ and we call $c_1 \cdots c_k$ the *predecessor sequence* of the symbol to be inserted into the bucket. We omit special buckets for any predecessor sequence containing \$-symbols from W and replace the character \$ and all following characters by As for the definition of the buckets, so we use only characters

$c_i \in \Sigma$ as predecessor symbols. The reason for this decision is that \$-symbols do not occur within a word W_i and thus, some level-k buckets, for example $A\$C$ for $k = 3$, would be empty by construction.

Like BCR [3] and ropeBWT2 [14], we can properly define the buckets of the final $BWT(W)$. For that purpose, we define $AW_j = (A^k \cdot W_j)$ as the word W_j prepended by k A symbols before the word W_j. We transfer the index i of each symbol in W_j to AW_j, so $AW_j[i] = W_j[i]$ for every word W_j. In Figure 1a, the grey As before the words W_j belong to AW_j. For each predecessor sequence $D \in \Sigma^k$, let $C_W(D) = \sum_{j=0}^{m-1} |\{i : M - |AW_j| - 1 \leq i < M \wedge AW_j[i, i+k-1] < D\}|$ be the number of predecessor sequences that are lexicographically smaller than D, obtained from the k-length substrings of AW_j. Then, we get that $BWT(W) = B_{A^k} B_{A^{k-1}C} B_{A^{k-1}G} B_{A^{k-1}T} B_{A^{k-2}CA} \cdots B_{T^{k-1}G} B_{T^k}$ is a well-defined partition into 4^k segments where a bucket is defined as $B_D[i] = BWT(W)[i + C_W(D)]$.

To properly insert the character $W_j[t]$ into its corresponding bucket, we need to know the predecessor sequence $D_j(t)$ for $W_j[t]$ in iteration t. Because the BWT can be obtained by taking the last column of the sorted rotations and the buckets are ordered, the predecessor sequence $D_j(t)$ occurs at the start of the rotation of the character $W_j[t]$, if the \$ is replaced with k A symbols. Thus, $D_j(t) = AW_j[t-1] \cdots AW_j[t-k]$ is the proper predecessor sequence for character $W_j[t]$.

Since $D_j(t) = AW_j[t-1] \cdots AW_j[t-k]$, $D_j(t)$ are the last k inserted characters of word W_j with the last inserted character $W_j[t-1]$ first or, if $t < k$, $D_j(t)$ is prepended by $k - t$ As. Especially, you can get $D_j(t+1) = W_j[t] \cdot D_j(t)[0, k-2]$ by shifting the previous predecessor sequence $D_j(t)$ and placing the last inserted character $W_j[t]$ at index 0. For example, in Figure 1b, $D_2(7) = CC$, because we insert $W_2[7-1] = C$ in the $CG = D_2(6)$ bucket, so $D_2(7) = W_2[6] \cdot D_2(6)[0, k-2] = C \cdot (CG)[0, 0] = CC$.

The idea is that knowing the bucket $B_{W_j[t-1]}$ to insert into and the insert position $P_j^{W_j[t-1]}(t)$ within that bucket is sufficient for insertion and that the global insert position $P_j(t)$ is not needed for insertion. To calculate the next insert position $P_j(t+1)$, we need $rank_{bwt(t)}(P_j(t), W_j[t])$ and $count_{bwt(t)}(W_j[t])$ for the LF-mapping. Using buckets and predecessor sequences, we can omit computing (or maintaining) and adding $count_{bwt(t)}(W_j[t])$: Let $P_j^{W_j[t-1]}(t)$ be the insert position of character $W_j[t]$ where we insert the character $W_j[t]$ into the bucket $B_{W_j[t-1]}$. Bauer et al. [3] prove $P_j^{W_j[t-1]}(t) = rank_{bwt(t-1)}(P_j(t-1), W_j[t-1])$ using a different notation. In case of $W_j[t] = A$, we get the next insert position $P_j^A(t+1) = rank_{bwt(t)}(P_j(t), A) + \alpha(t+1)$ due to our merge of the \$ bucket into the A bucket. In other words, $count_{bwt(t)}(W_j[t])$ is equal to the total number of characters in the buckets of smaller characters $c < W_j[t]$. If $W_j[t] = A$, $count_{bwt(t)}(A)$ is the number of active words $\alpha(t+1)$, which would be the size of the bucket $B_\$$ in $bwt(t+1)$ without the merge of the bucket $B_\$$ into the bucket B_A.

4.4 Trees

In order to save computation time for rank operations on previous buckets, we introduce four balanced binary trees, *A-tree*, *C-tree*, *G-tree*, and *T-tree*, of depth $2(k-1)$, as shown in Figure 1c for $k = 2$. We save the balanced binary trees in an array $T[0, 2^{2k} - 1]$ of length $4^k = \mathcal{O}(|W|)$ using the Eytzinger Layout: The root of the balanced binary trees, A-tree to T-tree, are saved in $T[4]$ to $T[7]$; the left child of a node $T[i]$ is stored at index $2i$ and the right child position at index $2i + 1$ in T. Each internal node has $|\Sigma| = 4$ counters, where the counter $T[i].c$ associated with $c \in \Sigma$ represents the number of characters c in the left subtree of node i. The leaf nodes are the buckets, which we discuss in Subsection 4.7.

The trees $T[4]$, $T[5]$, $T[6]$, and $T[7]$ represent the level-1 buckets B_A, B_C, B_G, and B_T, respectively. We continue to refer to each of these buckets as the concatenation of all level-k buckets of the corresponding tree.

We speed up the computation of $rank_{bwt(t)}(P_j(t), W_j[t])$ by storing additional information in $T[0]$ to $T[3]$. In $T[0]$, we store the total number of characters in the leaves of the A-tree. In $T[1]$, we store the number of characters of the C-tree plus the A-tree, so we have an accumulate count of the number of characters of the whole part of $bwt(t)$ prior to the start of the G-tree. In the same way, $T[2]$ and $T[3]$ represent the total number of characters up to the end of the G-tree and T-tree. Figure 1c visualizes the tree structure T.

We use $T[0]$ to $T[3]$ to determine the number of characters $W_j[t]$ up to the beginning of tree corresponding to $W_j[t-1]$. Thereby, $rank_{bwt(t)}(P_j(t), W_j[t])$ can be determined by processing only the tree corresponding to $W_j[t-1]$. We define the *level-1 rank* as $rank1(j,t) = rank_{B_{W_j[t-1]}}\left(P_j^{W_j[t-1]}(t), W_j[t]\right)$, which is the rank of symbol $W_j[t]$ inserted at its insert position $P_j^{W_j[t-1]}(t)$ on the tree of symbol $W_j[t-1]$.

$$rank_{bwt(t)}(P_j(t), W_j[t]) = \begin{cases} rank1(j,t) & D_j(t)[0] = A \\ T[0].(W_j[t]) + rank1(j,t) & D_j(t)[0] = C \\ T[1].(W_j[t]) + rank1(j,t) & D_j(t)[0] = G \\ T[2].(W_j[t]) + rank1(j,t) & D_j(t)[0] = T \end{cases}$$

4.5 Navigation

To get the local insertion position $P_j^{D_j(t)}(t)$ within level-k bucket $B_{D_j(t)}$, we navigate through the balanced binary trees in T using the predecessor sequence $D_j(t) = c_1 c_2 \dots c_k$: While the predecessor symbol c_1 of each word determines the selection of one of the 4 binary trees, the predecessor symbols $c_2, \dots, c_k$ of $W_j[t]$ determine the navigation direction inside the c_1-tree towards the bucket $B_{D_j(t)}$. Explicitly, for the navigation within the first two levels of the c_1-tree, the symbol c_2 is used, for the next two levels c_3, and so on. We navigate twice left if $c_i = A$; first left and second right if $c_i = C$; first right and second left if $c_i = G$; and twice right if $c_i = T$. If a left navigation step at node i is issued for $c = W_j[t]$, we need to increase $T[i].c$ by one, because we will insert the character c into a leaf in left subtree of node i.

If we want to insert symbols from different words at the same time and some symbols have to be inserted into a bucket left of an inner node i while other symbols have to be inserted into a bucket right of the inner node i, the left steps must increase $T[i].c$ at a node i before we can proceed with the right steps at node i. Otherwise, $T[i].c$ would not be the correct number of occurrences of the character c in the left subtree of node i.

For each right step, our goals are to transform the insert position $P_j^{W_j[t-1]}(t)$ into the local insert position $P_j^D(t)$, and to determine $rank1(j,t)$. The number of characters c in the local buckets before the bucket B_D is equal for all words of which characters are inserted into the bucket B_D. This equality is true both in the case of every single $c \in \Sigma$ and for the sum of all characters $c \in \Sigma$. We use accumulators $R_D.c$ to reduce the number of necessary additions in the right step: For each sequence D, $R_D.c$ is the sum of all counters $T[i].c$ where i is a node with a right step issued by D. Hereby, D must not be of length k during computation and can contain a $\leftarrow$ or $\rightarrow$ symbol at the end to indicate the step of the last half-processed predecessor symbol. We only compute R_D for predecessor sequences D where at least one character $W_j[t]$ has D as predecessor sequence in iteration t. Thus, we have at most $4 \cdot \min(4^k, m)$ different $R_D.c$ accumulators instead of exactly $4m$ if we would use four variables for each individual word.

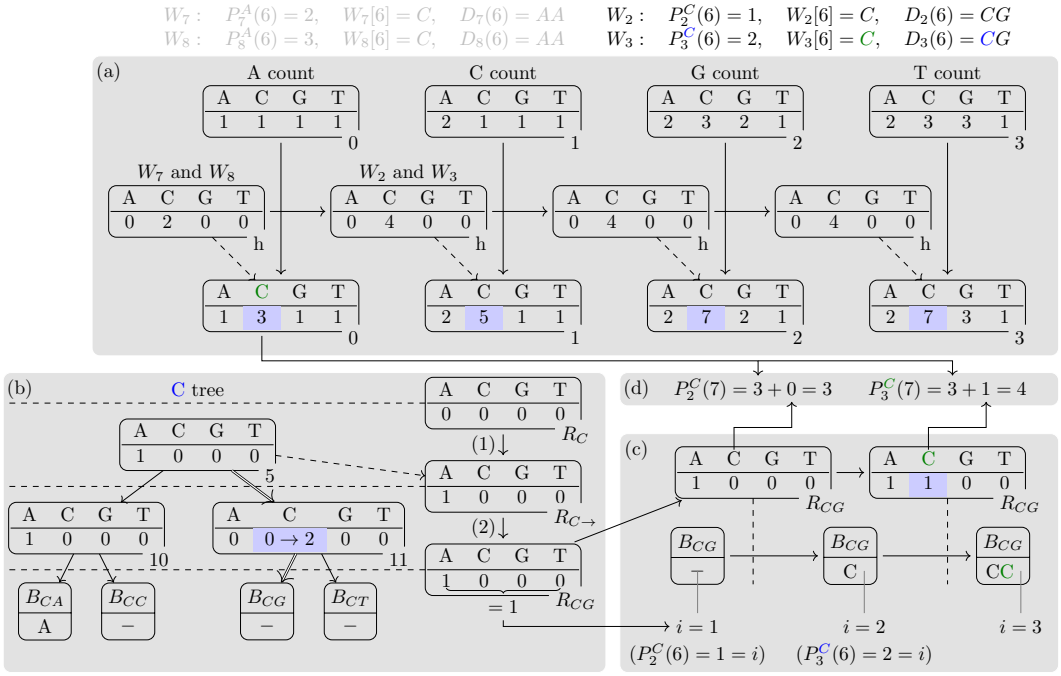


Figure 2 Insertion of $W_2[6]$ and $W_3[6]$ into IBB structure. Blue backgrounds mean that the numbers are changed. The green and blue C highlight which C is used in which location to determine the used tree or value.

Using the accumulator R_D for the character $W_j[t]$, so $D = D_j(t)$, the local insert position of $W_j[t]$ is $P_j^{D_j(t)}(t) = P_j^{W_j[t-1]}(t) - \sum_{c\in\Sigma} R_D.c$. Because $\sum_{c\in\Sigma} R_D.c$ is again constant for all words that insert into the bucket B_D, the sum $\sum_{c\in\Sigma} R_D.c$ is only computed once per predecessor sequence D. Furthermore, we do not subtract the sum $\sum_{c\in\Sigma} R_D.c$ from each $P_j(t)$. Instead, we initialize the position counter with $\sum_{c\in\Sigma} R_D.c$. An example for the insertion using $R_D.c$ is shown in Figure 2c.

To compute $rank1(j,t)$ for the insert position $P_j(t+1)$, we use $R_D.c$ with $c = W_j[t]$, which is the number of symbols c up to the bucket B_D. Then, we compute the *level-k rank* $rank\mathrm{k}(j,t) = rank_{B_{D_j(t)}}\left(P_j^{D_j(t)}(t), W_j[t]\right)$ during insertion into the bucket $B_{D_j(t)}$. Thus, we get the overall expression of the insert position $P_j^{W_j[t]}(t+1)$ for iteration $t+1$ from the iteration t as:

$$P_j^{W_j[t]}(t+1) = \begin{cases} \alpha(t+1) + R_{D_j(t)}.(W_j[t]) + rank\mathrm{k}(j,t) & D_j(t)[0] = A \\ T[0].(W_j[t]) + R_{D_j(t)}.(W_j[t]) + rank\mathrm{k}(j,t) & D_j(t)[0] = C \\ T[1].(W_j[t]) + R_{D_j(t)}.(W_j[t]) + rank\mathrm{k}(j,t) & D_j(t)[0] = G \\ T[2].(W_j[t]) + R_{D_j(t)}.(W_j[t]) + rank\mathrm{k}(j,t) & D_j(t)[0] = T \end{cases}$$

We parallelize the processing of the children each time both children need to be processed, as the further steps of the left and right child do not interfere with each other. We limit the number of threads close to the number of available processors on the used system, as this limit is most efficient.

Table 1 Used BWT construction algorithms.

approach	paper	implementation
IBB	this paper	`https://github.com/adlerenno/ibb`
BCR	[3]	`https://github.com/giovannarosone/BCR_LCP_GSA`
ropebwt	–	`https://github.com/lh3/ropebwt`
ropebwt2	[14]	`https://github.com/lh3/ropebwt2`
ropebwt3	–	`https://github.com/lh3/ropebwt3`
BigBWT	[4]	`https://gitlab.com/manzai/Big-BWT`
r-pfbwt	[22]	`https://github.com/marco-oliva/r-pfbwt`
grlBWT	[7]	`https://github.com/ddiazdom/grlBWT`
eGap	[8]	`https://github.com/felipelouza/egap`
gsufsort	[18]	`https://github.com/felipelouza/gsufsort`
divsufsort	[11]	`https://github.com/y-256/libdivsufsort`

4.6 Sort Words

If symbols from multiple words are inserted into the same bucket B_D, we must insert them in correct relative order. We insert the symbols from left to right; for that purpose, we sort the words W_j from which the characters $W_j[t]$ are taken according to their predecessor sequence $D_j(t)$ and then according to their local insert position $P_j^{D_j(t)}(t)$ in bucket $B_{D_j(t)}$. The resulting sort order is equal if we sort according to the first predecessor symbol $D_j(t)[0] = W_j[t-1]$ first and then to the insert position $P_j^{W_j(t-1)}(t)$. We will use the latter ones.

In iteration 0, all new started words are sorted by their insert position $P_j(0)$ by construction; thus they are also sorted according to their local insert position $P_j^{A^k}(0)$ because $P_j(0) = P_j^A(0) = P_j^{A^k}(0)$. Next, at the start of iteration t, we prepend all new active words W_z to the sorted list of active words, because their predecessor sequence D are A^k and their insert positions $P_z^A(t)$ are lower than the insert positions $P_j^A(t)$ of the older active words W_j due to the addition of $\alpha(t+1)$.

For iteration $t+1 > 0$, we sort the active words from iteration t at the end of iteration t by a single stable radix-sort step on $W_j[t]$. In Figure 1b, the radix sort sorts the words according to the characters $W_j[6]$ resulting in the same order of active words in iteration $t = 7$. The next insert positions $P_j(t+1)$ for one character $W_j[t+1]$ depends only on $rank_{bwt(t)}(P_j(t), W_j[t])$ for a group all words W_j with the same character c at position t, or on the constant term $\alpha(t+1)$ in case of $W_j[t] = A$. As the *rank* is a monotonic non-decreasing function of the position, the sort-order of the group of all W_j with $W_j[t] = c$ is stable. Thus, a single radix-sort step on $W_j[t]$ is sufficient to achieve the correct order of the active words W_j of the iteration t.

4.7 External Storage of Level-k Buckets

We store each level-k bucket B_D on external memory and insert into the bucket B_D by alternating two files. We denote the two files by $F_0(B_D)$ and $F_1(B_D)$. Let $L_D(t) \in \{0,1\}$ be the index of the file $F_{L_D(t)}(B_D)$ that we suppose to write to. Initially, we set $L_D(0) = 0$ for all D. If no word W_j has D as predecessor sequence, we skip the bucket B_D. Otherwise, we input $F_{1-L_D(t)}(B_D)$ as a stream. We output the symbols of the input stream to $F_{L_D(t)}(B_D)$ until we reach the next lowest insert position $P_j^D(t)$ of a word W_j. We then output $W_j[t]$ before continuing with the input stream and the next lowest insert position $P_z^D(t)$ of some

Table 2 Used datasets from NCBI, GAGE, or PacBio. We used partDNA [2] with parameter $h = 4$ to partition the datasets TAIR10, GRCh38, GRCm39, and JAGHKL01 into sets of words. $avg(|W_i|)$ is the average length of words W_i; M is the length of the longest word in $(W_i)_{0 \leq i < m}$. We visualize the distribution of word lengths of GRCh38 in Figure 3a.

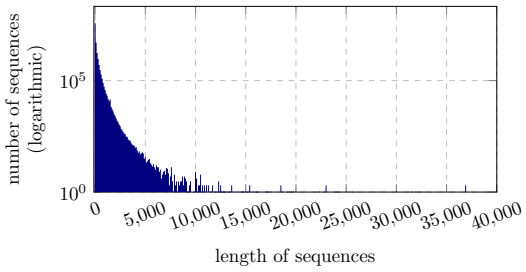
dataset	$avg(\|W_i\|)$	M	m	characters total
TAIR10	59	4,558	2,031,079	119,482,427
GRCm39	83	31,851	31,890,467	2,654,621,783
GRCh38	67	36,931	46,529,667	3,049,315,783
JAGHKL01	121	65,807	118,749,573	14,314,496,836
SRR11092057	137	151	5,314,809	727,671,874
influenza	1590	2,867	817,587	1,300,007,157
HGChr14	99	101	18,610,954	1,841,778,722
zymowastewater	143	150	73,996,930	10,563,509,439

word W_z. If no such word W_z with D as predecessor sequence exists, we output the rest of the input stream. Both, the input and output stream, are buffered for performance reasons. At the end, we flip $L_D(t)$ for the next iteration $t + 1$ if and only if bucket B_D was changed.

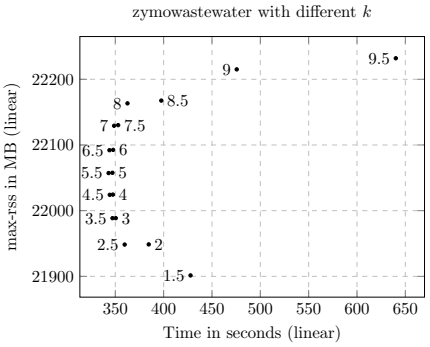
To get $rank\mathrm{k}(j, t)$ for the next insert position $P_j^{W_j[t]}(t + 1)$, we count the number of characters $W_j[t]$ that are written to file $F_{L_D(t)}(B_D)$ until the current local insert position $P_j^D(t)$. As $rank\mathrm{k}(j, t)$ is added to the value of $R_D.c$ with $c = W_j[t]$, we can directly increment the counter $R_D.c$ during writing c to the output file $F_{L_D(t)}(B_D)$. Thereby, we get $rank1(j, t)$ from $R_D.c$ with $c = W_j[t]$ just before outputting the symbol $W_j[t]$ at position $P_j^D(t)$.

In Figure 2, the characters $W_2[6]$ and $W_3[6]$ are inserted into $bwt(5)$ in iteration 6. First, in Figure 2a, we update $T[0]$ to $T[3]$ by increasing $T[0].C$ by 2 and $T[1].C$, $T[2].C$, and $T[3].C$ by 4 by using an accumulator h with four counters $h.c$. Second, in Figure 2b, we process the C-tree, because the first predecessor symbol is $D_2(6)[0] = D_3(6)[0] = C$. We do a right and then a left child step because the second predecessor symbol is $D_2(6)[1] = D_3(6)[1] = G$. In Figure 2c, the right step is marked with (1) and the left step is marked with (2). We initialize the accumulators $R_C.c$ with 0 for all c. Because there is no word that requires a left child step at $T[5]$, we omit cloning R_C into $R_{C\leftarrow}$ and just keep R_C as $R_{C\rightarrow}$. Additionally, we add the counters of the root of the C tree at $T[5]$ to $R_{C\rightarrow}$ to R_C, so we properly have the number of characters in the left subtree of $T[5]$ in R_C. Then, we process the right child $T[2 * 5 + 1] = T[11]$. First, we increase for W_2 and W_3 $T[11].C = 0$ twice by 1 to 2, because both are left steps. As no word requires a right step at $T[11]$, we can omit that branch. By navigating to the child with index $2 * 11 = 22$, we reach a leaf of the tree, thus we insert the characters $W_2[6]$ and $W_3[6]$ to that bucket B_{CG}.

In Figure 2c, we perform the insertion into the bucket CG. The position counter i is initialized with value 1, which is the sum of the accumulators $R_C.c$. Because $P_j^C(6) = 1 = i$, we insert the character $W_2[6] = C$ at the beginning of bucket B_{CG}. We increase $R_{CG}.C$ by 1 for the inserted C of word W_2. In Figure 2d, we compute the next insert position $P_2^C(7)$ for W_2 before incrementing $R_{CG}.C$. The next insert position $P_2^C(7) = 3$ for W_2 is the sum of the number of C before the C-tree $T[5]$, which is $T[0].C = 3$, and the number of C within the C tree up to that position, which is $R_{CG}.C = 0$. In the same way we insert $W_3[6]$ and get the next insert position $P_3^C(7) = 4$ for W_3 in iteration $t = 7$.



(a) Distribution of the sequence length of the partitioned GRCh38 file. We summarize sequences over an interval of one-hundred characters. 91.26% of the 46,529,667 sequences are shorter than 200 characters, but there are two sequences between 36,900 and 37,000 characters.



(b) Construction time and RAM consumption of IBB for zymowastewater dataset tested for $1.5 \leq k \leq 9.5$.

Figure 3 Length distribution and parameter evaluation.

5 Experimental Results

We compare the BWT construction algorithms of Table 1 on the datasets listed in Table 2 regarding construction time and RAM usage. We obtain time, maximum resident set size (max-rss), amount of input (io_in) and amount of output (io_out) from Snakemake [20].[3] We test two scenarios: First, to obtain comparative results regarding construction time to the in-memory algorithms, we run all external-memory algorithms on RAM disks. Second, we compare only the external-memory algorithms without using RAM disks. Ambiguous bases in the datasets are omitted. We performed all tests on a Debian 5.10.209-2 machine with 128GB RAM and 32 Cores Intel(R) Xeon(R) Platinum 8462Y+ @ 2.80GHz.

We pass the number of 32 threads explicitly to IBB, grlBWT, r-pfbwt, BigBWT, and ropebwt3; other approaches, like grlBWT, that parallelize their construction, determine the number of threads on their own, because they have no such command line option. We pass the option `-R` to ropebwt, ropebwt2 and ropebwt3 to avoid processing the read sequences in both directions. We use the parameter options `w1 = 10`, `p1 = 100`, `w2 = 5`, and `p2 = 11` for r-pfbwt as suggested. Also, we do not force eGap into external, semi-external or internal memory mode.

We tested BCR using the BCR_LCP_GSA implementation that supports datasets of unequal length, nevertheless BCR failed on all datasets. BCR is an external algorithm designed for equal length datasets: The implementation extends shorter input sequences to have the same length as the longest input sequence, which is necessary for its representation in external memory. In the case of TAIR10, this adds 9,138,175,655 characters which is more than 76 times the input of 119,482,427 characters.

GrlBWT failed using the RAM disk and we could not determine the cause for these failures. Thus, the time and RAM usage for grlBWT was obtained without using the RAM disk. BigBWT and eGap are external algorithms as well, but their implementations lack an option to change the directory of temporary files; thus, we obtain their times and RAM usages without using the RAM disk.

[3] The test is available at `https://github.com/adlerenno/ibb-test`.

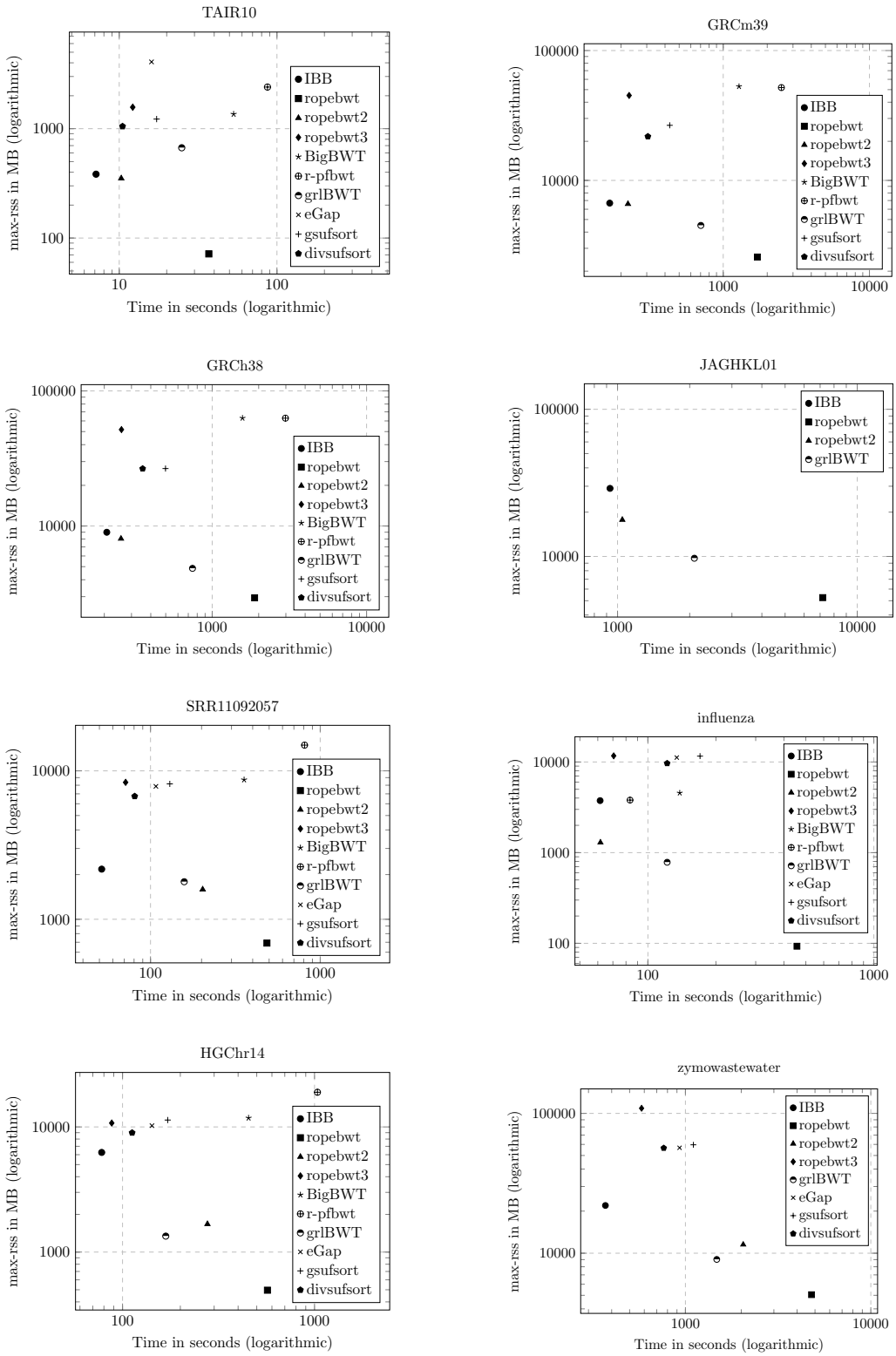


Figure 4 BWT construction times and maximum resident set sizes (max-rss). A missing point means that the construction algorithm aborts or does not create an output file.

Our implementation of IBB is able to perform only one of the two levels of the last predecessor symbol, so more values for k are possible. We refer to these by $x.5$ values for the parameter k.[4] We use $k = 2.5$ in all our tests as parameter; in Figure 3b, we show an evaluation for parameter k. The fastest time IBB achieves on the dataset zymowastewater is 343.0701 seconds for $k = 5.5$. From 2.5 to 7.5 all recorded times only differ by 17 seconds.

The scatter plots in Figure 4 show the time in seconds and RAM usage in MB given the file and the approach. We say an algorithm A is superior to another approach B if A uses less time, which means the point of A is left of the point of B, and less RAM, so the point of A is lower than the point of B. For example, ropebwt is superior to BigBWT on the file TAIR10, because ropebwt uses less time and less RAM compared to BigBWT, so ropebwts point is in the left of and below the point of BigBWT.

Our test shows that IBB is the fastest construction algorithm on all datasets. On the influenza dataset, which has longer but fewer sequences, ropebwt2 and IBB both take 61 to 62 seconds, with IBB being slightly faster. On the other datasets, IBB is between 11.33% (HGChr14) and 36.16% (zymowastewater) faster than the second fastest algorithm. The second fastest algorithms are divsufsort on zymowastewater, ropebwt3 on SRR11092057, and ropebwt2 on the remaining datasets. Using the time-optimal parameter $k = 5.5$ from Figure 3b, IBB is 40.85% faster than divsufsort on zymowastewater.

Ropebwt uses the lowest amount of RAM on all datasets followed by either ropebwt2 or grlBWT. The latter two mostly offer pareto-optimal space-time tradeoffs for BWT construction. They are always followed by IBB, which is then superior to any other tested approach in both, construction time and RAM usage.

In Table 5 in Appendix A, we compare the time and external memory used by the external-memory algorithms without using a RAM disk. Of the algorithms in Table 1, BCR, BigBWT, eGap, grlBWT, r-pfbwt, and IBB are external-memory algorithms. We analyzed the total amount of input and output written by the algorithms, obtained as the sum of the io_in and io_out value. IBB has a much higher amount of input and output; nevertheless, except for the GRCm39 and JAGHKL01 datasets, IBB is the fastest external-memory approach.

6 Conclusion

We presented IBB, a BWT construction algorithm designed for length-diverse DNA datasets. IBB employs right alignment, tree-based data structures, fine buckets, and one-step radix sorting. Our experiments show that IBB outperforms existing state-of-the-art BWT construction algorithms by being 10% to 40% faster on most datasets while maintaining competitive memory consumption.

References

1 Enno Adler, Stefan Böttcher, Rita Hartel, and Cederic Alexander Steininger. IBB. Software, swhId: `swh:1:dir:42d4aa3c09d658e12540f4090749c8c3d7966642` (visited on 2025-06-30). URL: `https://github.com/adlerenno/ibb`, `doi:10.4230/artifacts.23784`.

2 Enno Adler, Stefan Böttcher, and Rita Hartel. String Partition for Building Long BWTs, 2024. `arXiv:2406.10610`.

[4] The IBB implementation actually takes $2k$ as input.

3 Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.*, 483:134–148, 2013. doi:10.1016/J.TCS.2012.02.002.

4 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms Mol. Biol.*, 14(1):13:1–13:15, 2019. doi:10.1186/S13015-019-0148-5.

5 Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. In *Digital SRC Research Report*. Citeseer, 1994.

6 Jasbir Dhaliwal, Simon J. Puglisi, and Andrew Turpin. Trends in Suffix Sorting: A Survey of Low Memory Algorithms. In *Thirty-Fifth Australasian Computer Science Conference, ACSC 2012, Melbourne, Australia, January 2012*, pages 91–98, 2012. URL: http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV122Dhaliwal.html.

7 Diego Díaz-Domínguez and Gonzalo Navarro. Efficient construction of the BWT for repetitive text using string compression. *Inf. Comput.*, 294:105088, 2023. doi:10.1016/J.IC.2023.105088.

8 Lavinia Egidi, Felipe A. Louza, Giovanni Manzini, and Guilherme P. Telles. External memory BWT and LCP computation for sequence collections with applications. *Algorithms Mol. Biol.*, 14(1):6:1–6:15, 2019. doi:10.1186/S13015-019-0140-0.

9 Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight Data Indexing and Compression in External Memory. In *LATIN 2010: Theoretical Informatics, 9th Latin American Symposium, Oaxaca, Mexico, April 19-23, 2010. Proceedings*, pages 697–710, 2010. doi:10.1007/978-3-642-12200-2_60.

10 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

11 Johannes Fischer and Florian Kurpicz. Dismantling DivSufSort. In *Proceedings of the Prague Stringology Conference 2017, Prague, Czech Republic, August 28-30, 2017*, pages 62–76, 2017. URL: http://www.stringology.org/event/2017/p07.html.

12 Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature methods*, 9(4):357–359, 2012.

13 Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology*, 10:1–10, 2009.

14 Heng Li. Fast construction of FM-index for long sequence reads. *Bioinform.*, 30(22):3274–3275, 2014. doi:10.1093/BIOINFORMATICS/BTU541.

15 Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinform.*, 26(5):589–595, 2010. doi:10.1093/BIOINFORMATICS/BTP698.

16 Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009. doi:10.1093/BIOINFORMATICS/BTP336.

17 Felipe A. Louza, Simon Gog, and Guilherme P. Telles. Induced Suffix Sorting for String Collections. In *2016 Data Compression Conference, DCC 2016, Snowbird, UT, USA, March 30 - April 1, 2016*, pages 43–52, 2016. doi:10.1109/DCC.2016.27.

18 Felipe A. Louza, Guilherme P. Telles, Simon Gog, Nicola Prezza, and Giovanna Rosone. gsufsort: constructing suffix arrays, LCP arrays and BWTs for string collections. *Algorithms Mol. Biol.*, 15(1):18, 2020. doi:10.1186/S13015-020-00177-Y.

19 Ruibang Luo, Jeanno Cheung, Edward Wu, Heng Wang, Sze-Hang Chan, Wai-Chun Law, Guangzhu He, Chang Yu, Chi-Man Liu, Dazong Zhou, et al. MICA: A fast short-read aligner that takes full advantage of Many Integrated Core Architecture (MIC). *BMC bioinformatics*, 16:1–8, 2015.

20 Felix Mölder, Kim Philipp Jablonski, Brice Letcher, Michael B. Hall, Christopher H. Tomkins-Tinch, Vanessa Sochat, Jan Forster, Soohyun Lee, Sven O. Twardziok, Alexander Kanitz, Andreas Wilm, Manuel Holtgrewe, Sven Rahmann, Sven Nahnsen, and Johannes Köster. Sustainable data analysis with snakemake [version 2; peer review: 2 approved]. *F1000Research*, 10(33), 2021. doi:10.12688/f1000research.29032.2.

21 Ge Nong, Sen Zhang, and Wai Hong Chan. Two Efficient Algorithms for Linear Time Suffix Array Construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011. doi:10.1109/TC.2010.188.

22 Marco Oliva, Travis Gagie, and Christina Boucher. Recursive Prefix-Free Parsing for Building Big BWTs. In *Data Compression Conference, DCC 2023, Snowbird, UT, USA, March 21-24, 2023*, pages 62–70, 2023. doi:10.1109/DCC55655.2023.00014.

23 Simon J. Puglisi, William F. Smyth, and Andrew Turpin. A taxonomy of suffix array construction algorithms. In *Proceedings of the Prague Stringology Conference, Prague, Czech Republic, August 29-31, 2005*, pages 1–30, 2005. URL: http://www.stringology.org/event/2005/p1.html.

24 Jared T Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome research*, 22(3):549–556, 2012.

A List of Symbols

Symbol	Explanation
S	A single string.
Σ	The alphabet for strings.
$rank_S(x, c)$	Number of occurrences of c in $S[0, x-1]$.
$count_S(c)$	Number of characters smaller than c in S.
$BWT(W)$	Burrows-Wheeler transform of W.
$(S_i)_{0\leq i<m}$	Collection of m words.
j, z	Indices used for words.
M	Maximal length of words in the collection of words.
W	String with $\$_i$ to define the BWT of a collection.
t	Current iteration (counter).
$LF(i)$	Last-to-First mapping of position i. Used to reverse the Burrows-Wheeler transform.
$bwt(t)$	Partial BWT after iteration t.
$(W_i)_{0\leq i<m}$	Collection of words obtained from $(S_i)_{0\leq i<m}$ by reversing and right aligning.
$P_j(t)$	Insert position of word W_j in iteration t. We sometimes call this insert position global to make the difference to the other insert positions more verbose.
$\alpha(t)$	Number of active words at iteration t.
$SB(t)$	Bitvector at iteration t. $SB(t)[i] = 1$ if and only if word W_i is active.
B_c	Level-1 bucket for predecessor symbol c.
k	Parameter to IBB. Defines the depth of the trees as $2k-2$ and the length of the predecessor sequence.
$B_{c_1 \dots c_k}$	Level-k bucket for predecessor sequence $c_1 \dots c_k$
AW_j	Constructed word that includes the prefixed A^k. $AW_j = (A^k \cdot W_j)$
D	Any predecessor sequence. Typically of length k, yet not exclusively.
$C_W(D)$	Predecessor sequence counting based on AW_j and length k.
$D_j(t)$	Predecessor sequence of word W_j in iteration t.
$P_j^c(t)$	Insert position of word W_j in iteration t relative to the bucket B_c for a single character c.
$P_j^{D_j(t)}(t)$	Insert position of word W_j in iteration t relative to the bucket $B_{D_j(t)}$ for predecessor sequence $D_j(t)$.
T	Array to save the tree.

Symbol	Explanation
$rank1(j,t)$	Rank of inserted character $W_j[t]$ at position $P_j^{W_j[t-1]}(t)$ on the string $B_{W_j[t-1]}$, which is the level-1 bucket, so the whole string within the $W_j[t-1]$ tree.
$R_D.c$	Accumulators during tree processing. D is the predecessor sequence (not necessarily of length k), $c \in \Sigma$.
$rank\mathrm{k}(j,t)$	Rank of inserted character $W_j[t]$ at position $P_j^{D_j(t)}(t)$ on the string $B_{D_j(t)}$, which is the level-k bucket.
$F_i(B_D)$	File with index i associated to the bucket B_D.
$L_D(t)$	Index of file to write of bucket B_D in iteration t.
i, x	Indices with minimal scope.

B Evaluation Results

Table 4 The following tables list the results for the tests using RAM disk.

(a) TAIR10.

Approach	Time (s)	max-rss (MB)
BCR	–	–
BigBWT	53.1195	1,357.90
divsufsort	10.4938	1,049.59
eGap	16.0234	4,078.12
grlBWT	24.9048	669.00
gsufsort	17.2666	1,223.87
IBB	7.1295	383.66
r-pfbwt	87.1017	2,401.18
ropebwt	36.9868	71.68
ropebwt2	10.3008	350.90
ropebwt3	12.1775	1,570.97

(b) GRCm39.

Approach	Time (s)	max-rss (MB)
BCR	–	–
BigBWT	1,282.7032	52,893.75
divsufsort	305.4121	21,796.86
eGap	–	–
grlBWT	703.2101	4,501.62
gsufsort	430.7094	26,563.23
IBB	167.4545	6,685.82
r-pfbwt	2,499.6059	51,853.26
ropebwt	1,712.1969	2,569.87
ropebwt2	223.1947	6,585.71
ropebwt3	227.3670	45,098.62

(c) GRCh38.

Approach	Time (s)	max-rss (MB)
BCR	–	–
BigBWT	1,571.8187	62,954.50
divsufsort	354.5520	26,577.39
eGap	–	–
grlBWT	745.3975	4,857.22
gsufsort	498.4121	26,630.89
IBB	207.5513	8,989.36
r-pfbwt	2,981.6389	62,886.01
ropebwt	1,882.8899	2,938.64
ropebwt2	256.6377	8,056.41
ropebwt3	258.3919	51,735.07

(d) JAGHKL01.

Approach	Time (s)	max-rss (MB)
BCR	–	–
BigBWT	–	–
divsufsort	–	–
eGap	–	–
grlBWT	2,091.7559	9,722.14
gsufsort	–	–
IBB	930.3123	29,011.98
r-pfbwt	–	–
ropebwt	7,177.5101	5,251.80
ropebwt2	1,046.8844	17,736.01
ropebwt3	–	–

(e) SRR11092057.

Approach	Time (s)	max-rss (MB)
BCR	–	–
BigBWT	356.1337	8,709.25
divsufsort	80.6930	6,741.88
eGap	107.6240	7,862.65
grlBWT	158.0011	1,790.23
gsufsort	129.9375	8,169.07
IBB	51.5733	2,176.63
r-pfbwt	808.8667	14,901.21
ropebwt	485.2979	691.24
ropebwt2	203.0154	1,588.63
ropebwt3	71.3488	8,364.67

(f) influenza.

Approach	Time (s)	max-rss (MB)
BCR	–	–
BigBWT	138.3262	4,552.87
divsufsort	121.6796	9,660.54
eGap	134.2194	11,183.71
grlBWT	121.6891	783.07
gsufsort	170.1249	11,595.05
IBB	61.4567	3,747.00
r-pfbwt	83.3559	3,797.16
ropebwt	456.7642	92.92
ropebwt2	61.6558	1,294.27
ropebwt3	70.5649	11,686.07

(g) HGChr14.

Approach	Time (s)	max-rss (MB)
BCR	–	–
BigBWT	453.5586	11,790.17
divsufsort	112.1837	8,980.93
eGap	142.3352	10,236.23
grlBWT	167.9500	1,341.46
gsufsort	171.9143	11,349.88
IBB	77.8972	6,255.21
r-pfbwt	1,035.8426	18,928.85
ropebwt	568.6316	495.94
ropebwt2	276.8750	1,673.65
ropebwt3	87.8484	10,737.50

(h) zymowastewater.

Approach	Time (s)	max-rss (MB)
BCR	–	–
BigBWT	–	–
divsufsort	763.8530	56,533.53
eGap	929.7517	56,763.04
grlBWT	–	–
gsufsort	1,103.9456	59,621.83
IBB	370.2614	21,948.47
r-pfbwt	–	–
ropebwt	4,788.0798	5,051.52
ropebwt2	2,049.8674	11,528.41
ropebwt3	–	–

Table 5 The following tables list the results for external-memory approaches without using a RAM disk.

(a) TAIR10.

Approach	Time (s)	IO (MB)
BCR	–	–
BigBWT	54.7431	255.72
eGap	16.7038	0.17
grlBWT	24.9048	1,355.56
IBB	14.7806	15,064.05
r-pfbwt	92.4510	338.87

(b) GRCm39.

Approach	Time (s)	IO (MB)
BCR	–	–
BigBWT	1,371.5356	6,586.37
eGap	447.5178	2,130.45
grlBWT	703.2101	39,075.34
IBB	449.2580	511,059.49
r-pfbwt	2,714.2478	12,496.36

(c) GRCh38.

Approach	Time (s)	IO (MB)
BCR	–	–
BigBWT	1,572.0601	7,696.68
eGap	570.0827	2,952.55
grlBWT	745.3975	60,962.59
IBB	493.8601	557,734.03
r-pfbwt	3,275.8786	20,418.35

(d) JAGHKL01.

Approach	Time (s)	IO (MB)
BCR	–	–
BigBWT	–	–
eGap	–	–
grlBWT	2,091.7559	185,232.58
IBB	2,512.7515	4,235,386.38
r-pfbwt	–	–

(e) SRR11092057.

Approach	Time (s)	IO (MB)
BCR	–	–
BigBWT	355.0525	1,952.04
eGap	109.0181	624.06
grlBWT	158.0011	7,629.49
IBB	66.8494	34,861.29
r-pfbwt	847.7592	3,129.21

(f) influenza.

Approach	Time (s)	IO (MB)
BCR	–	–
BigBWT	148.4186	1,834.74
eGap	139.4808	970.80
grlBWT	121.3235	3,362.62
IBB	66.0257	23,235.80
r-pfbwt	95.4392	445.43

(g) HGChr14.

Approach	Time (s)	IO (MB)
BCR	–	–
BigBWT	462.9071	2,586.88
eGap	147.4293	402.06
grlBWT	167.9500	10,243.43
IBB	91.0889	45,861.17
r-pfbwt	1,119.1047	3,431.50

(h) zymowastewater.

Approach	Time (s)	IO (MB)
BCR	–	–
BigBWT	–	–
eGap	933.8026	5,264.78
grlBWT	1,475.5007	92,775.28
IBB	619.8390	443,232.04
r-pfbwt	–	–