Title

Efficiently Merging r-indexes

Permalink

https://escholarship.org/uc/item/8086b9pc

Authors

Oliva, Marco

Rossi, Massimiliano

Sirén, Jouni

et al.

Publication Date

2021-03-26

DOI

10.1109/dcc50243.2021.00028

Copyright Information

Efficiently Merging r-indexes

Marco Oliva*, Massimiliano Rossi*, Jouni Sirén†, Giovanni Manzini‡,
Tamer Kahveci*, Travis Gagie§ and Christina Boucher*

*University of Florida
Gainesville, FL, USA

§Dalhousie University
Halifax, Canada

‡University of Eastern Piedmont
Alessandria, Italy

†University of California Santa Cruz
Santa Cruz, CA, USA

Abstract

Large sequencing projects, such as GenomeTrakr and MetaSub, are updated frequently (sometimes daily, in the case of GenomeTrakr) with new data. Therefore, it is imperative that any data structure indexing such data supports efficient updates. Toward this goal, Bannai et al. (*TCS*, 2020) proposed a data structure named *dynamic* r-index which is suitable for large genome collections and supports incremental construction; however, it is still not powerful enough to support substantial updates. Here, we develop a novel algorithm for updating the r-index, which we refer to as RIMERGE. Fundamental to our algorithm is the combination of the basics of the *dynamic* r-index with a known algorithm for merging Burrows-Wheeler Transforms (BWTs). As a result, RIMERGE is capable of performing batch updates in a manner that exploits parallelism while keeping the memory overhead small. We compare our method to the dynamic r-index of Bannai et al. using two different datasets, and show that RIMERGE is between 1.88 to 5.34 times faster on reasonably large inputs.

1 Introduction

The 1,000 Genomes Project [1] was largely completed in 2012 and the 100,000 Genomes Project milestone was reached in 2018 [2]. Read alignment tools — such as BWA [3], Bowtie [4, 5] and SOAP [6] — have been fundamental to the analysis of these datasets. They take as input a set of sequence reads and one or more reference genome(s), build an index from the reference genomes, and use this index to align the reads to the genome(s) allowing a small number of insertions and deletions. Read aligners use the FM-index [7] to store the indexed genomes in a compressed form. The FM-index, which consists of the *Burrows-Wheeler transform* (BWT) of the input text, a rank data structure over the BWT and the suffix array (SA) sampled at constant size intervals, cannot effectively scale to terabyte sized datasets because the space used by SA samples grows linearly with the input size. In an effort to build an index in sub-linear

MO, MR, and CB are funded by the National Science Foundation (NSF) IIS (Grant No. 1618814), NSF IIBR (Grant No. 2029552) and National Institutes of Health (NIH) NIAID (Grant No. HG011392 and R01AI141810). MO and TK are funded by US NAVY (Grant No. N62473-18-2-0011). TG is funded by NSF IIBR (Grant No. 2029552) and NIH NIAID (Grant No. HG011392) and NSERC Discovery Grant RGPIN-07185-2020. JS was supported by the National Human Genome Research Institute (Grant No. 1U01HG010961-01, 1R01HG010485-01 and 1U41HG010972-01). GM was partially supported by PRIN (Grant No. 2017WR7SHH) and by INdAM-GNCS (Project 2020MFAIS-IoT). CB is funded by NSF SCH: INT (Grant No. 2013998).

space, Gagie et al. [8] made a significant breakthrough by defining a variant of the FM-index whose space requirements is proportional to the number of runs r of the BWT. Hence, they refer to their index as the r-index. Compared to the FM-index, the r-index is able to locate all occurrences of a pattern in a text of length n in optimal time using $\mathcal{O}(r \log(n/r))$-space. This is possible since the SA samples are stored only at the beginning and at the end of each run of the BWT.

Conceptually the result of Gagie et al. was a significant step forward, but it did not lead to efficient construction of the r-index. This was only later achieved by Boucher et al. [9] and then Kuhnle et al. [10], where they developed Big-BWT, an algorithm that is able to construct the r-index for very large datasets. It is based on a preprocessing step where a scan of the text generates a dictionary and a parse that are, in turn, used to build the BWT and the SA samples. The construction space of the BWT and the SA samples is proportional to the size of the dictionary and parse. This result proved to be extremely useful in bioinformatics where the data is highly repetitive.

Although Big-BWT is very effective to build the r-index for large datasets, every time a new sequence is included in the dataset, the r-index has to be rebuilt. This overhead may be affordable when the dataset is small, but it is impractical when the size of the update is only a small fraction of the original index. For this problem, Sirén [11] and Ferragina et al. [12] proposed solutions for the FM-index which succinctly merges two BWTs. Later refined in [13], the BWT merge algorithm by Sirén is based on the idea of computing the rank of each suffix extracted from one BWT with respect to the other. Those ranks are then used to guide the merge of the characters of the two BWTs. Prezza and Rosone [14] recently proposed an alternative approach to merge two BWTs. Their solution consists in building the Document Array, an array that will tell us whether the i-th entry of the output BWT comes from the first or the second BWT in input. While these two approaches solve efficiently the problem of merging two BWTs they do not address the problem of merging two r-indexes.

Recently, Bannai et al. [15] proposed a dynamic version of the r-index. It maintains a data structure to compute the LF-mapping and a balanced search tree for the predecessor search on the SA samples. Both these operations are performed in $\mathcal{O}(\log r)$ time. Hence, the backward search in the dynamic r-index takes $\mathcal{O}(\log r)$-time per character. Even though their method, denoted as *dynamic* r-index, allows to incrementally build the index for the dataset, in practice its construction takes much more time than Big-BWT.

The ability to update an index significantly extends its practicality. Being able to store only the index without the need to store the uncompressed file for future updates makes the database more manageable. Moreover, as in the case of the 1,000 Human Genomes project, databases are updated as projects proceed. Updating the index allows us to take advantage of the information contained in the new sequences promptly.

In this paper, we focus on updating the r-index without relying on dynamic data structures. We devise and implement an algorithm for efficiently merging two or more r-indexes, which we refer to as RIMERGE. We formally define the steps of RIMERGE, and prove that it correctly builds the r-index for the desired input. Intuitively RIMERGE operates in two steps. It starts computing the position where every suffix extracted from the second r-index should be inserted in the first r-index and, simultaneously, computing the candidate SA samples for the output index. Once the first step is completed it proceeds interleaving the two indexes using the information computed previously.

We demonstrate the utility of our method by reporting the time, space and memory needed to update an r-index built from two different genetic datasets: 1,512 haplotypes of chromosome 19 from the 1,000 Genomes Project [1], and 7,048 strains of salmonella from GenomeTrakr [16]. We show that although *dynamic* r-index is faster on small inputs (less than 8 haplotypes of chromosome 19, and less than 32 strains of salmonella), RIMERGE quickly becomes faster than the *dynamic* r-index. RIMERGE was almost two times faster for 16 haplotypes and over five times faster for 256 haplotypes. RIMERGE is implemented in C++, open source and available at `https://github.com/marco-oliva/rimerge`.

2 Background

In this section, we introduce the notation and concepts that we will use in the paper. We assume the reader is familiar with basic index based data structures, including self-balancing search trees, and refer the reader to Navarro [17] for a further exposition of these concepts.

Merging BWTs Given two texts $S[1..n]$ and $T[1..m]$ terminated by $\$_S < \$_T$ respectively, we want to compute the BWT of the text $ST = S[1]..S[n]T[1]..T[n]$ as the result of the merge of BWT_S and BWT_T. As described in [11], the first step to merge the two BWTs is to compute the rank of each suffix of T in the set of sorted suffixes of S. This can be done by backward searching T in BWT_S [18]. The key observation is the following. Given an index $i > 1$ and the rank k_i of the i-th suffix of T, $T[i..m]$, we can compute the rank k_{i-1} of the $(i-1)$-th suffix of T by counting how many suffixes of S, smaller than $T[i..m]$ are preceded by $T[i-1]$. This can be accomplished using the LF-mapping, i.e. $k_{i-1} = \mathsf{LF}_S(k_i, T[i-1])$.Here we use the notation $\mathsf{LF}(i, c)$ to specify the character we are computing the LF-mapping on, therefore $\mathsf{LF}(i) = \mathsf{LF}(i, \mathsf{BWT}[i])$. We store the ranks of all suffixes of T in an array called *rank array* RA. The ranks are stored in suffix array order of the suffixes of T. Formally, we define the *rank array* $\mathsf{RA}[1..m]$ as follows, for all $i = 1, \ldots, m$, $\mathsf{RA}[i] = |\{j \mid S[j..n] \leq T[SA_T[i]..m]\}|$. To compute the rank array during the backward search of T in BWT_S, we reconstruct T using the LF-mapping. Since $\$_S < \$_T$, we set $\mathsf{RA}[1] = 1$ and provided we have computed $\mathsf{RA}[i]$, the value of the $LF_T[i]$-th entry of the rank array is computed as $\mathsf{RA}[LF_T[i]] = LF_S(\mathsf{RA}[i], BWT_T[i])$. One important property of the rank array is that it is monotonically non-decreasing, i.e. for all $1 \leq i < j \leq m$, $\mathsf{RA}[i] \leq \mathsf{RA}[j]$. The information contained in RA is then used to merge BWT_S with BWT_T. We insert, between the symbols $\mathsf{BWT}_T[i]$ and $\mathsf{BWT}_T[i+1]$, all symbols in $\mathsf{BWT}_S[\mathsf{RA}[i] + 1..\mathsf{RA}[i+1]]$. The merging algorithm is summarized in Algorithm 1, and Figure 1 shows how to use the RA to interleave two BWTs.

Algorithm 1 BWT Merge

```
Input: The BWTs of two texts $S[1..n]$ and $T[1..m]$.
Output: The BWT of the text $ST$.
procedure MERGE($\mathsf{BWT}_S, \mathsf{BWT}_T$)
    $i \leftarrow 1, \mathsf{RA}[1] \leftarrow 1$
    while $\mathsf{BWT}_T[i] \neq \$$ do
        $\mathsf{RA}[\mathsf{LF}_T[i]] \leftarrow \mathsf{LF}_S(\mathsf{RA}[i], \mathsf{BWT}_T[i])$
        $i \leftarrow \mathsf{LF}_T[i]$
    end while
    $i \leftarrow j \leftarrow k \leftarrow 1$
    while $i \leq m$ do
        while $j \leq RA[i]$ do
            $\mathsf{BWT}_{ST}[k] \leftarrow \mathsf{BWT}_S[j]$
            $k \leftarrow k+1, j \leftarrow j+1$
        end while
        $\mathsf{BWT}_{ST}[k] \leftarrow \mathsf{BWT}_T[i]$
        $k \leftarrow k+1, i \leftarrow i+1$
    end while
    while $j \leq n$ do
        $\mathsf{BWT}_{ST}[k] \leftarrow \mathsf{BWT}_S[j]$
        $k \leftarrow k+1, j \leftarrow j+1$
    end while
    return $\mathsf{BWT}_{ST}$
end procedure
```

Run-length encoded BWT and the *r*-index Given a text S and its BWT, a *run* in BWT is a maximal substring of equal characters in BWT. The run-length encoded BWT is an equivalent representation of the BWT where each run is represented as the character of the run and its length.

The r-index [8] is an evolution of the FM-index designed to better exploit the repetitiveness of the input which can be approximately measured by r, the number of runs in the text's BWT. The r-index stores the SA values in the positions corresponding to the beginning and the end of each run, and stores the BWT as a run-length encoded string. Therefore, the space needed is $\mathcal{O}(r)$. In order to compute the samples not stored with the index, the r-index uses a function that, given the value of the suffix array in position i, allows to compute the value of SA in position $i+1$. This function is implemented as a predecessor data structure that stores the SA samples at the end of each run. We refer to this function as ϕ.

Dynamic *r*-index Bannai et al. [15] presented a dynamic version of the r-index that is based on the following observation. Given a text S and its BWT, if we want to prepend a character c to S we can compute the rank of cS among the suffixes of S as $\mathsf{LF}_S(i,c)$, where i is the index of the end marker of S in the BWT. We then replace the end marker with c and insert a new end marker between $\mathsf{LF}_S(i,c)$ and $\mathsf{LF}_S(i,c)+1$. During this process we need to update the set of SA samples. Replacing the end marker with c may cause the join of two runs, while inserting the end marker in position $\mathsf{LF}_S(i,c)$ may split a run in the BWT. In the first case, we just need to remove the samples corresponding to the end marker and to the joined runs, while in the second case we need to insert three new samples: one corresponding to the end marker and the two samples at the beginning and at the end of the new runs generated by the split. To address this issue, we make use of the following lemma.

Lemma 2.1 ([15]) *Given a text $S[1..n]$, its BWT and the SA samples at the beginning of each run, if we know $j = \mathsf{SA}[k+1]$ for some position k, we can compute, for any character c, the text position j' such that $S[j'..n]$ is the lexicographically smallest* *suffix that is larger than $cS[SA[k]..n]$ (if such $S[j'..n]$ exists).*

The intuition behind Lemma 2.1 is to look at $\mathsf{BWT}[k+1..n]$ and whether it contains the character c or not. We have three cases. Either i) $\mathsf{BWT}[k+1] = c$ and thus, $j' = SA[k+1]-1$; or ii) $\mathsf{BWT}[k+1] \neq c$ and the first occurrence of c in $\mathsf{BWT}[k+1..n]$ is the beginning of a run, at which we have the corresponding SA sample stored so we can compute the value of j'; or iii) $\mathsf{BWT}[k+1] \neq c$ and $\mathsf{BWT}[k+1..n]$ does not contain the character c, so we look for the first occurrence of the smallest character $c' > c$ that occurs in S, which is also the beginning of a run at which we have the corresponding sample stored so we can compute the value of j'.

Symmetrically to Lemma 2.1, we can compute the lexicographicaly largest suffix smaller than $cS[\mathsf{SA}[k]..n]$ provided that we have the SA samples at the end of each run of the BWT.

Lemma 2.2 *Given a text $S[1..n]$, its BWT and the SA samples at the beginning of each run, if we know $j = \mathsf{SA}[k]$ for some position k, we can compute, for any character c, the text position j' such that $S[j'..n]$ is the lexicographically largest suffix that is smaller than $cS[SA[k]..n]$ (if such $S[j'..n]$ exists).*

3 Merging r-indexes

A simple way to merge two r-indexes is to merge the BWTs and then, as a second step, reconstruct the missing SA samples and discard those not needed. The missing samples could be reconstructed using the predecessor data structure and the function ϕ of the two texts S and T. However, this trivial approach is time consuming since the number of steps that may be necessary is proportional to the average length of a run in the r-indexes.

Our algorithm avoids the worst-case complexity of the trivial algorithm by taking advantage of the construction of the rank array to build a set of candidate samples. In order to present our algorithm we first describe how to: i) reconstruct the missing samples from S, and ii) reconstruct the missing samples from T. Both sets of samples are computed during the construction of the rank array. In the following, we first describe T because it is the more immediate given that the SA samples are available during the backward search. We note that since we do not need the predecessor data structure, we store only the run-length encoded BWT [19] and the SA samples at the beginning and at the end of each run.

3.1 Computing the Missing Samples of T

The missing samples correspond in BWT_{ST} to characters in the middle of a run that become the beginning or the end of a run. In the case of T, this may occur when two elements in a run have different rank values in RA or when a character from T is placed in a a run of S of a different character. While we extract all the suffixes of T from BWT_T to compute their rank relative to S, we also compute the associated suffix array values. For each distinct value of RA, the only positions where we could break a run in BWT_T are the first and the last occurrences of the current value of RA. All the suffix array samples corresponding to runs ends/starts between the first and the last occurrence are also samples of BWT_T and no suffix of S will be placed in this interval.

We make use of two self-balancing search trees $\mathcal{T}_\ell$ and $\mathcal{T}_r$. The self-balancing search tree $\mathcal{T}_\ell$ (resp. $\mathcal{T}_r$) stores the values of SA_T corresponding to the position of the first (resp. last) occurrence of each distinct value of the rank array RA. Since we want to compute $\mathcal{T}_\ell$ and $\mathcal{T}_r$ while we compute the rank array, we store in addition the value of the position in which each suffix array value occurs. Thus, for each new value of RA, we keep the suffix array value associated with the smallest (resp. largest) position computed so far.

3.2 Computing the Missing Samples of S

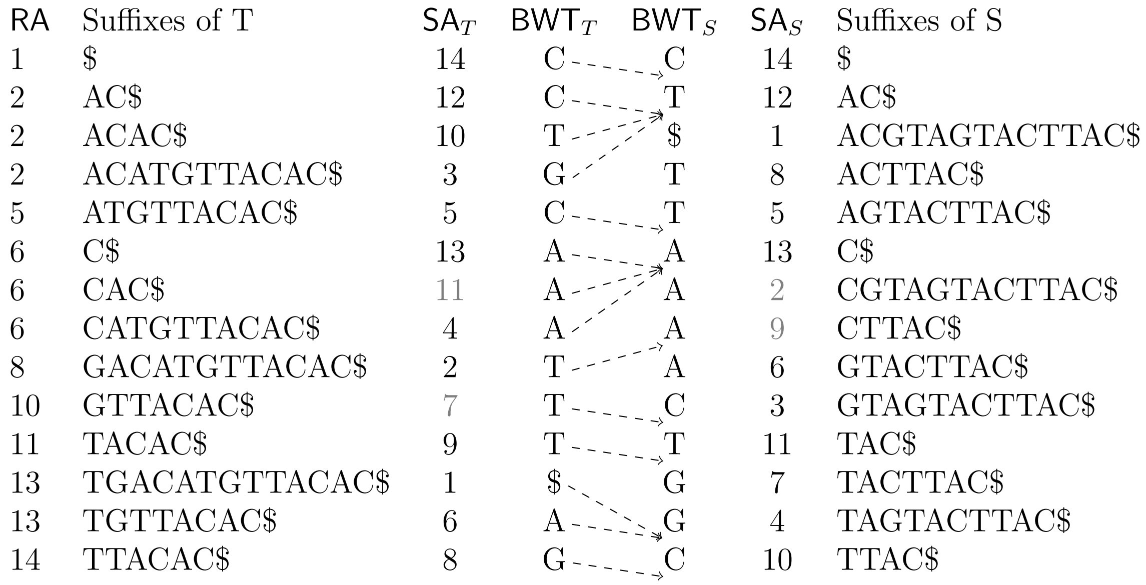
We need to reconstruct a sample of S when a character c of BWT_T is placed between two characters of BWT_S different from c. We check for this condition while computing the rank array values. We use Lemmas 2.1 and 2.2 to compute the missing samples of the suffix array of S. As well as for the missing samples of T, for S we define one self-balancing search tree $\mathcal{S}$ which stores, for each distinct value of RA, the values of the suffix array of S in positions corresponding to the value of RA and the same value incremented by one. Since we use Lemma 2.1 to compute the missing samples of S, we require that the second value of SA_S is sampled. An example of the merge of two r-indexes is shown in Figure 1.

3.3 RIMERGE*: Putting it all together*

Given two texts $S[1..n]$ and $T[1..m]$ over an alphabet Σ, terminated by $\$_S < \$_T$ respectively, their BWTs and the suffix array samples at the beginning and at the end of each run of their BWT. We assume without loss of generality that $m < n$, otherwise we exchange the role of S and T.

As the first step, we compute the rank array of T with respect to S. We recover the text T from BWT_T backward, starting from $T[m-1]$, i.e. $\mathsf{BWT}_T[1]$. Each time a new rank array value is computed, we update the data structures storing candidate samples for the merged BWT described in the previous section.

As second step, we use the rank array and the data structures with the candidate samples to build BWT_{ST} and its SA samples. For each element $\mathsf{RA}[i]$ of the rank array, we write all the elements of BWT_S with an index smaller than or equals to $\mathsf{RA}[i]$ and greater than $\mathsf{RA}[i-1]$. If the first element written from BWT_S is the beginning of a run in BWT_{ST}, we need to store the suffix array samples of the end of the previous run and the beginning of the current run. The sample corresponding to the end of the previous run is stored in $\mathcal{T}_r$ in position $\mathsf{RA}[i-1]$, while we compute the sample for the beginning of the current run as follows. If the current element of BWT_S is either the beginning or the end of a run in BWT_S, its corresponding suffix array sample is stored in SA_S, otherwise the corresponding suffix array sample is stored in $\mathcal{S}$ in position $\mathsf{RA}[i-1]$. When writing the i-th character of BWT_T, if this character is the



RA	Suffixes of T	SA_T	BWT_T	BWT_S	SA_S	Suffixes of S
1	$	14	C	C	14	$
2	AC$	12	C	T	12	AC$
2	ACAC$	10	T	$	1	ACGTAGTACTTAC$
2	ACATGTTACAC$	3	G	T	8	ACTTAC$
5	ATGTTACAC$	5	C	T	5	AGTACTTAC$
6	C$	13	A	A	13	C$
6	CAC$	11	A	A	2	CGTAGTACTTAC$
6	CATGTTACAC$	4	A	A	9	CTTAC$
8	GACATGTTACAC$	2	T	A	6	GTACTTAC$
10	GTTACAC$	7	T	C	3	GTAGTACTTAC$
11	TACAC$	9	T	T	11	TAC$
13	TGACATGTTACAC$	1	$	G	7	TACTTAC$
13	TGTTACAC$	6	A	G	4	TAGTACTTAC$
14	TTACAC$	8	G	C	10	TTAC$

$\mathsf{BWT}_{ST} =$	C	C	T	C	T	G	$	T	T	C	A	A	A	A	A	A	T	A	C	T	T	T	G	G	$	A	C	G
$\mathsf{SA}_{ST} =$	14	28	12	26	24	17	1	8	5	19	13	27	25	18	2	9	16	6	3	21	11	23	7	4	15	20	10	22
																*				*								

Figure 1: Merge of the r-indexes of S = ACGTAGTACTTAC$ and T = TGACATGTTACAC$. From left to right, RA represents the rank array: the next column reports the suffixes of T in lexicographic order; SA_T and BWT_T are the suffix array and the BWT of T; BWT_S and SA_S are the BWT and the suffix array of S; The last column reports the suffixes of S in lexicographic order. The arrows drawn from BWT_T to BWT_S show the rank of each character of BWT_T with respect to the characters of BWT_S. At the bottom BWT_{ST} reports the BWT resulting from the merge of S and T as well as SA_{ST} reports the suffix array. The elements of SA_T, SA_S, and SA_{ST} that are not sampled in the r-index are colored in gray. The asterisks identify suffixes that are sampled in BWT_{ST} but not in BWT_T and BWT_S. In particular 9 is a missing sample of S, while 21 is a missing sample of T.

beginning of a run in BWT_{ST}, we store the suffix array samples of the end of the previous run and the beginning of the current run. The sample corresponding to the end of the previous run is either stored in $\mathcal{S}$ in position $\mathsf{RA}[i]$ or in SA_S, while the sample for the beginning of the current run is stored in $\mathcal{T}_\ell$ in position $\mathsf{RA}[i]$.

3.4 Analysis

The computation of the rank array takes $\mathcal{O}(m)$ steps, one for each suffix of T. At each step we update the set of candidate samples stored in a red-black tree which requires $\mathcal{O}(\log m)$-time per operation. Hence, the overall time needed for the first step of the algorithm is $\mathcal{O}(m \log m)$. Interleaving the two BWTs, given the information computed in the first step, takes $\mathcal{O}(n + m)$ time. The computation of the SA samples requires $\mathcal{O}(r)$ queries to a red-black tree, where r is the number of runs in BWT_{ST}. In total, the time required to merge two r-indexes is $\mathcal{O}(n + m + m \log m + r \log m)$.

The rank array uses $\mathcal{O}(m)$ words and the red-black tree requires $\mathcal{O}(m \log m)$ words. The LF-mapping and the SA samples take $\mathcal{O}(r)$-space. Thus, the total amount of memory used by the algorithm is $\mathcal{O}(m \log m + r)$.

4 Experiments and Discussion

To evaluate the performances of RIMERGE we compare it against the implementation of the r-index of Bannai et al. [15] (*dynamic* r-index). We performed our experiments using two distinct datasets. The first one consists in a set of 1,512 chromosome 19 haplotypes extracted from the VCF file of phase-3 of the 1,000 Genomes Project [1]. The second contains 7,048 strains of salmonella from GenomeTrakr [16]. The size of the dataset containing all the sequences for chromosome 19 and salmonella is 84GB and 34GB, respectively.

All experiments were performed on an Intel(R) Xeon(R) CPU E5-2698 v3 at 2.30GHz with 32 cores and 125GB of RAM. The machine had no other significant CPU tasks running, and up to 32 threads of execution were used. The given time statistics were recorded with the linux `/usr/bin/time` measurement tool while the memory usage statistics were retrieved with the `malloc_count` tool (`https://github.com/bingmann/malloc_count`).

Using the chromosome 19 data, we built the r-index for the first 1,000 haplotypes using Big-BWT. Next, we used RIMERGE to insert new sequences from the remaining 512. The increments size were 1, 2, 4, 8, 16, 32, 64, 128, 256 and 512. The same has been done with the salmonella dataset. We have built the r-index for the first 5,000 strains and then used RIMERGE to insert 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1,024 and 2,048 strains. Since the *dynamic* r-index does not offer an option to serialize the index we built the index for the first 1,000 chromosome 19 haplotypes and then we recorded the insertion time and memory for each sequence inserted. We applied the same strategy on salmonella, starting from 5,000 strains.

We report the wall clock time in seconds of both methods in Table 1, along with the division of the total time of RIMERGE in building and merging. The total wall clock time for both methods is plotted in Figure 2. RIMERGE quickly became faster for updating the r-index. The *dynamic* r-index was faster on small inputs (less than 16 haplotypes of chromosome 19 and 32 strains of salmonella), RIMERGE proved to be faster than the *dynamic* r-index on both the datasets. The improvement in the speed was more substantial when the number of inserted sequences was larger. For example, when 16 haplotypes were inserted, RIMERGE was 1.88 times faster, when 128 haplotypes were inserted the improvement in the speed more than doubled to 4.84, and when 256 and 512 haplotypes were inserted, the improvement increased even more. In particular when inserting 512 new chromosome 19 haplotypes, as well as when inserting 2,048 strains of salmonella, it was five times faster. Moreover, the difference between the two methods when inserting a small number of sequences was negligible; less than 300 seconds when inserting up to 32 strains of salmonella and less than 500 seconds when inserting up to 8 chromosome 19 haplotypes. The memory consumption of both methods was negligible; neither exceeded 37 GB even on the largest input.

Finally, we point out that RIMERGE needs as many bytes of disk space as the size of the text we want to insert in the index to store the rank array RA while the *dynamic* r-index stores its data structures only in memory.

No. Sequences	Update Size (GB)	RIMERGE Build (s)	RIMERGE Update (s)	RIMERGE Total (s)	*dynamic* r-*index* (s)
1,001	0.06	47	472	519	79
1,002	0.12	51	492	542	158
1,004	0.24	54	511	566	316
1,008	0.47	63	539	602	632
1,016	0.94	78	592	670	1,264
1,032	1.89	112	634	747	2,528
1,064	3.78	166	1,010	1,176	5,056
1,128	7.56	282	1,803	2,085	10,112
1,256	15.12	482	3,306	3,788	20,224
1,512	30.24	897	6,771	7,669	40,448

Table 1: Comparison of the wall clock time required by RIMERGE against the *dynamic* r-index on the chromosome 19 dataset. We report the number of sequences in the index ("No. Sequences"), size in GB of the data to be inserted ("Update Size"), the total time for RIMERGE ("Total") and the total time of *dynamic* r-index. In addition, for RIMERGE we report separately the time required for building the update text's index with Big-BWT ("RIMERGE Build") and the time required to merge the update's index ("RIMERGE Update"). Since the initial set consisted in 1,000 haplotypes the number of sequences inserted is the number reported in "No. Sequences" minus 1,000.

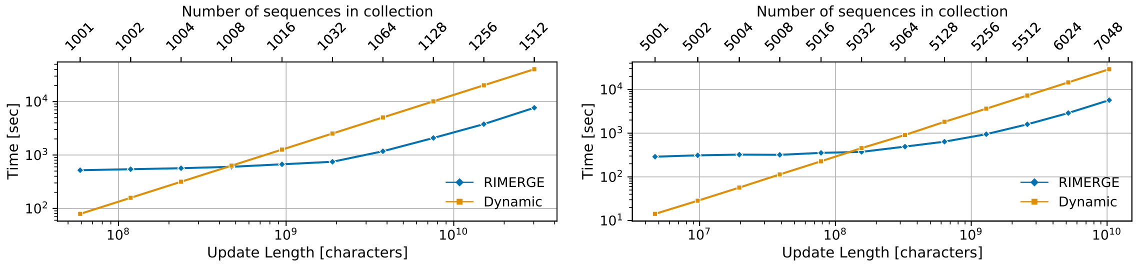


Figure 2: Comparison of the wall clock time for updating the index of 1,000 chromosome 19 haplotypes with up to 512 sequences (left) and updating the index of 5,000 strains of salmonella with up to 2,048 sequences (right).

5 Conclusions

In this work we address the problem of adding new sequences to a pre-existing r-index. We show how to maintain the r-index structure computing the necessary samples without relying on the ϕ function. We show that our algorithm, making use of the rank array RA, can perform batch updates that turn out to be faster than single insertions. In terms of memory consumption, RIMERGE differs from the *dynamic* r-index by a small amount (less than 10GB). A possible extension of this work consists in adding the support for deleting sequences since the same theoretical results can be applied to perform deletions as well. As discussed in the previous sections, the RA values guide the interleaving of the two indexes. The same information can be used to mark the sequences that we want to remove. To support deletions, during the interleave step, instead of inserting the characters from the second index we simply need to remove them.

References

[1] The 1000 Genomes Project Consortium, "A global reference for human genetic variation," *Nature*, vol. 526, pp. 68–74, 2015.

[2] C. Turnbull et al., "The 100,000 genomes project: bringing whole genome sequencing to the NHS," *Br. Med. J.*, vol. 361, 2018.

[3] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows–Wheeler Transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[4] B Langmead, C Trapnell, M Pop, and S Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome Biol.*, vol. 10, pp. R25, 2009.

[5] B. Langmead and S.L. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nat. Methods*, vol. 9, no. 4, pp. 357–359, 2012.

[6] R. Li et al., "De novo assembly of human genomes with massively parallel short read sequencing," *Genome Res.*, vol. 20, no. 2, pp. 265–272, 2010.

[7] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proc. of FOCS*, 2000, pp. 390–398.

[8] T. Gagie, G. Navarro, and N. Prezza, "Fully functional suffix trees and optimal text searching in BWT-runs bounded space," *J. ACM*, vol. 67, no. 1, pp. 1–54, 2020.

[9] C. Boucher, T. Gagie, A. Kuhnle, B. Langmead, G. Manzini, and T. Mun, "Prefix-free parsing for building big BWTs," *Algorithms Mol. Biol.*, vol. 14, pp. 13, 2019.

[10] A. Kuhnle, T. Mun, C. Boucher, T. Gagie, B. Langmead, and G. Manzini, "Efficient construction of a complete index for pan-genomics read alignment," in *Proc. of RECOMB*, 2019, pp. 158–173.

[11] J. Sirén, "Compressed suffix arrays for massive data," in *Proc. of SPIRE*, 2009, pp. 63–74.

[12] P. Ferragina, T. Gagie, and G. Manzini, "Lightweight data indexing and compression in external memory," in *Proc. of LATIN*, 2010, pp. 697–710.

[13] J. Sirén, "Burrows-Wheeler transform for terabases," in *Proc. of DCC*, 2016, pp. 211–220.

[14] N. Prezza and G. Rosone, "Space-efficient computation of the LCP array from the Burrows-Wheeler transform," in *Proc. of CPM*, 2019, p. 7:1–7:18.

[15] H. Bannai, T. Gagie, and T. I, "Refining the r-index," *Theor. Comput. Sci.*, vol. 812, pp. 96–108, 2020.

[16] E.L. Stevens et al., "The public health impact of a publically available, environmental database of microbial genomes," *Front. Microbiol.*, vol. 8, pp. 808, 2017.

[17] G. Navarro, *Compact Data Structures: A Practical Approach*, Cambridge University Press, 2016.

[18] W.-K. Hon, T.W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yiu, "A space and time efficient algorithm for constructing compressed suffix arrays," *Algorithmica*, vol. 48, no. 1, pp. 23–36, 2007.

[19] V. Mäkinen and G. Navarro, "Succinct suffix arrays based on run-length encoding," *Nord. J. Comput.*, vol. 12, no. 1, pp. 40–66, 2005.