Abstract—The block-sorting data compression method of Burrows-Wheeler(BWT) has received considerable attention in anticipation that it may be comparable, or even superior, to the Ziv-Lempel codes. Among many variants of BWT, the Schindler transform(ST) adopts a sorting scheme with restricted key length to greatly boost the transform speed. In this paper, we unify the forward and backward ST transforms by introducing a set of properties of the limited order context. The unified framework not only facilitates the hardware implementation of ST based data compression scheme, but also brings out faster and more memory efficient algorithms to compute the backward ST.

I. INTRODUCTION

The Burrows and Wheeler Transform(BWT)[1] can reorder a text into a more “compressible” sequence. The BWT itself does not reduce the length of a given text; however, the transformed text tends to group identical characters together so that the probability of finding repeated characters is increased substantially. As a consequence, the transformed text could be better compressed by fast locally-adaptive encoding algorithms, such as run-length-encoding and move-to-front coding following by Huffman or arithmetic coders.

The BWT and its variants have received considerable attention in anticipation that it may outperform Ziv-Lempel codes in string algorithms. It particularly attracts attention because of its simple and elegant operation and excellent practical compression performance[2], [3], [4], [5]. This has also stimulated research on exploring their underlying mathematical theories[6], [7], [8], [9], [10], [11].

Sorting all suffix of the input string into a lexicographical order constitutes a major computational bottleneck of the BWT. A comparison-based algorithm, which requires $\Omega(n \log n)$ comparisons, may take $\Omega(n^2 \log n)$ time for suffix sorting, and analogously a normal radix sorting algorithm may take $\Omega(n^2)$ time. Linear time suffix sorting can be achieved by building a suffix tree and obtaining the sorted order from its leaves[12], [13], [14], [6]. However, a suffix tree involves overhead, particularly in space requirements, which commonly makes it too expensive to use for suffix sorting alone. An alternative solution is the suffix array method introduced by Manber and Myers [15]. Suffix array is an array containing the start positions of all the suffix in a lexicography order.

The methods to construct suffix array can be divided into three types: direct comparison algorithms, doubling algorithms and linear algorithms. Direct comparison algorithms, in their simplest form, sort suffixes one character at a time as quicksort does. This technique was applied by Burrows and Wheeler[1], and has since been significantly improved by Seward[16] and Manzini and Ferragina[17]. Direct comparison algorithms are $O(n^2 \log n)$ in the worst case, but tend to be very fast for real inputs. Doubling algorithms on the other hand, with each sorting pass, double the depth to which the suffixes are sorted. In this way, the suffixes are sorted in a logarithmic number of passes, giving an overall worst-case time bound of $O(n \log n)$. Manber and Myers[15] were the first to apply this idea to suffix sorting and their approach was later significantly improved by Larsson and Sadakane [18], [19]. Three different algorithms recently discovered require only $O(n)$ time in the worst case to compute the suffix array of a given string on an indexed (integer) alphabet[20], [21], [22]. However, after a thorough performance study on the algorithms built on different suffix data structures carried out by Puglisi, Smyth and Turpin [23], confirms that in general, the doubling algorithms[19] not only are much faster than the suffix tree based methods but also outperform the linear algorithms.

Another approach for speeding up the transformation is through reducing the problem size by sorting only a portion of the matrix. This solution was due to Schindler and has been known as the Schindler Transform(ST)[7], [8], [24], [25], [26], [27]. The ST is built upon the concept of limited and unlimited order contexts, which says that if the last character of a text is viewed as a symbol, then its preceding characters will be viewed as its contexts. The context of a character can have different limited orders raging from 0 to $N-1$, while the full text is deemed as the unlimited order context. The ST then resembles the BWT but with the sorting part of the matrix limited to the $k$-order($k \in [0, N]$)contexts only. The reduced comparison length accelerates the transformation.

A major tradeoff caused by the ST’s partial sorting scheme is reflected in the backward transform component: the backward ST is more complicate than the backward BWT. This is because although each of the unlimited context is unique,
the uniqueness of any limited-order context considered by the partial sorting scheme in ST can no longer be guaranteed. To deal with this situation, Schindler proposed a hash based approach where the text retrieval has to rely on a hash table based context lookup, which in turn has to rely on the complete retrieval of all the k-order contexts [26]. As a result, the backward ST proposed by Schindler has an \(O(kN)\) time/space complexity. Schindler’s algorithm and its combinatorial properties are explained more thoroughly by Yokoo[24].

Observing that neither the full restoration of contexts nor the hash-based context lookup is required by the backward BWT, Nong and Zhang have proposed an auxiliary vectors based framework to eliminate the use of hash technique[28], and achieve a space complexity of \(O(N)\). They then further developed the algorithm framework to achieve a time complexity of \(O(N \log_2 k)\)[29], by employing a “doubling technique” that appeared in fast suffix sorting algorithms [18], [15].

In this paper, we first explore a set of properties about the \(k\)-order contexts to compute the backward ST. Based on these properties, we show that the forward ST and the backward ST can actually be fit into a unified algorithm framework. We then propose an algorithm to compute both forward and backward ST with a time complexity of \(O(N \log k)\) and a space complexity of \(O(N)\). This algorithm will show that, the solutions of both the forward and the backward ST problems can be essentially the same.

The rest of this paper is organized as follows. Section II introduces some related works which would help to explain our idea in the subsequent sections. Section III revisits current implementation technologies for the BWT and the forward and backward procedures of ST. Our main results for unifying the forward and backward ST are given in section III, where we first discuss several properties and theorems, and based on them we then present an algorithm framework to unify the two processes. Finally we discuss the advantages and disadvantages of our framework in section IV.

II. RELATED WORKS

We first revisit some related ideas about BWT and ST, and introduce a set of notations and definitions that will be used in the rest of this paper.

A. The Burrows-Wheeler Transform and Its Backward

A text \(S\) is a size-\(N\) row vector \([x_1 \ x_2 \ x_3 \ \cdots \ x_{N-1} \ s]\), where \(x_i\) is a character of an ordered alphabet \(\Sigma\). At the end of the vector there is a special character \(s \in \Sigma\) larger than any other character in \(\Sigma\) and can only appear at the end of a text. We define \(S[i]\) a predecessor\(\text{(successor)}\) of \(S[j]\), if \(i < j\)\((i > j)\), and a immediate predecessor\(\text{immediate successor)}\) if \(j = i+1\).The BWT over the text \(S\) is a reversible permutation of \(S\), which is conducted in the following three steps. First, a \(N \times N\) matrix \(OM\) is derived from the input \(S\): the first row is \(S\); and the remaining rows are constructed by applying successive cyclic left-shifts to \(S\). Second, we sort all the row vectors in \(OM\) into a lexicography order, and obtain a new matrix \(M\). Finally, we extract and output the last column of \(M\), and denote it by \(L\). As a result, \(L\) is the output of BWT, and we write \(BWT(S) = L\). As an example, we show in Fig.1 the result of the BWT on the text "mississippi" (this sample input text will be used throughout the paper for illustration purpose).

The backward BWT aims to restore the original test \(S\) from \(L\). The procedure is conducted in the following three steps: First, recover the first column(denoted by \(F\)) of \(M\) by sorting \(L\) into the alphabetic order of \(\Sigma\), assuming that the sorting algorithm is a stable one. Its correctness can be easily derived from the definition of \(M\). Second, construct a mapping \(P\) which maps the position of every character in \(L\) to its position in \(F\). Note that, due to the stable sorting strategy, \(P(i) < P(j)\), when \(1 \leq i < j \leq N\) and \(L[i] = L[j]\). Finally, we denote the position of \(s\) in \(L\) by \(INDEX_s\), then the text \(S\) can be reconstructed as \(FP^{N}(INDEX_s) \cdots FP^{P(INDEX_s)} F^{P(INDEX_s)}\), where \(P^{N}(INDEX_s) = P^{N-1}(INDEX_s)\) and \(P^{1}(INDEX_s) = P^{0}(INDEX_s)\), This is because, \(F(P[i])\) exactly the immediate predecessor of \(F[i]\) in \(S\), which has been proven in [1].

B. The Schindler Transform and Its Backward

The ST adopts a two-hierarchy sorting scheme, i.e. the lexicographical sorting criterion in tandem with the positional sorting criterion. Specifically, firstly, the \(k\)-order ST will sort all the rows of \(OM\) according to their \(k\)-order context, which is defined to be the initial \(k\) characters of each row. Then if there are any two identical \(k\)-order contexts, the tie will be resolved by preserving the relative order between them in the original \(OM\). When a \(k\)-order ST is mentioned, we use a subscript version of the above symbols, e.g. \(M_k\), \(L_k\), \(P_k\).

For example, if we apply the 2-order ST algorithm to "mississippi", we will obtain \(M_2\) as illustrated in Fig.2 and the output \(ST_2(S) = [s \ m \ s \ p \ s \ i \ s \ i \ i \ s]\).

The backward ST proposed by Schindler involves two phases: the \(k\)-order contexts retrieval and the original text retrieval. the \(k\)-order contexts retrieval involves a time and space complexity of both \(O(kN)\). Due to the partial sorting strategy of ST, the relation that \(F(P[i])\) is exactly the immediate predecessor of \(F[i]\) in \(S\) no longer holds. This is because,
in BWT, when a character $F[i]$ is mapped to the character $F[P(i)]$, the mapping actually finds the whole immediate predecessor row of the $i$th row in $M$; but, the same mapping in the ST can only find the $k$-order context of the immediate predecessor row for the $i$th row. Therefore, we cannot use $P$ to correctly find out the true immediate predecessor row for a given row among multiple candidate rows that share exactly the same $k$-order context. To address this challenge, Schindler proposed a hashing-based strategy in [26], which is outlined in Fig.3.

Observing that neither the full restoration of contexts nor the hash-based context lookup is required by the backward BWT, Nong and Zhang have proposed an auxiliary vectors based framework, named Generalized Backward BWT(GIBWT), to eliminate the use of hash technique[28]. The framework is described in Fig.4.

The correctness of the framework is based on the following property:

Property 1: For $k \in [1, N]$, if $M_k[i, 1 : k] = M_k[j, 1 : k]$ and $1 \leq i < j \leq N$, $L_k[j]$ must be a successor of $L_k[i]$ in $S$ unless $i$ is $Index_8$. This property is due to the ST’s positional sorting criterion. The criterion states that if two rows share the same $k$-order context, their relative order in the original $OM$ matrix must remain the same in $M_k$. Consequently, the relative order of the last characters of the two rows in $OM$ must be preserved in $M_k$. Therefore, if we use the above backward recovery strategy, the two rows sharing the same $k$-order context must be visited in the reverse order. The exception occurs when $i = Index_8$; in this special case, $L_k[i]$, the last character of the original text, will always be the first character to be recovered. For this purpose, the two vectors $T_k$ and $C_k$ are introduced. We define a maximal sequence of adjacent rows in $M_k$ which have the same $k$-order context a group. $T_k$ is used to record the index of the starting row of each group, and $C_k$ is a length vector recording the length of each group. The Line 13-21 in Fig.4 utilize the above property to recover the original text $S$. This step requires only $O(N)$ time. Line 5-12 describe the method to build $T_k$ and $C_k$ using a vector $D_k$, which maintains the border information of the groups. $D_k[i]$ is 1 if the $i$th row
in $M_k$ is the first row in a group, otherwise is 0.

The complexity of the above framework is determined by the complexity of the procedure of MakeD in Line 4 of Fig.4, which builds the vector $D_k$. The original proposed algorithm[28] achieves a space complexity of $O(N)$ and a time complexity of $O(kN)$. By employing the idea of doubling technique, they then further developed an algorithm with time complexity $O(N \log k)$ and space complexity $O(N)[29]$.

III. UNIFYING FORWARD AND BACKWARD ST

A. A Unified Framework

As already mentioned in Section I, sorting all the suffixes of a string $S$ into lexicographical order is the major component of the ST. We will show that the suffix sorting algorithms can also be used to compute the backward ST, thus form the heart of our unified framework.

Firstly, we introduce the definition of cycle, which is the foundation for developing our unified framework. We denote resulting vector of stably sorting $L_k$ as $F_k$ and the backward mapping of $P_k$ as $Q_k$. Thus $Q_k$ is the mapping the position of each character in $F_k$ to its position in $L_k$.

**Definition 1:** Cycle $\alpha_i$: A sequence of characters $[L_k[i], L_k[Q_k(i)], \ldots L_k[Q_k^{\alpha_i}(i)]]$, satisfying
1. $Q_k^{\alpha_i+1}(i) = i$; 2. if $m, n \in [1, l]$ and $m \neq n$, then $Q_k^m(i) \neq Q_k^n(i)$, where $l$ is the length of $\alpha_i$, and $Q_k^0(i) = Q_k(i)$.

Therefore, cycle $\alpha_i$ is a list of characters consisting of a subset of the characters in $L_k$, starting from $L_k[i]$, and following by $L_k[Q_k(i)], L_k[Q_k(Q_k(i))], \ldots$, ending just before another occurrence of $L_k[i]$.

The following 2 properties about $\alpha_i$ and the $k$-order contexts of $M_k$ is the basis of our framework:

**Property 2:** Given $L_k$ and $Q_k$, $M_k[i, 1 : k] = [L_k[Q_k(i)], L_k[Q_k(Q_k(i))], \ldots L_k[Q_k^{\alpha_i}(i)]]$.

**Property 3:** The $k$-order context of row $i$ in $M_k$ is the first $k$ characters of the string made up of the unlimited repetitions of cycle $\alpha_{Q_k(i)}$.

Property 2, can be easily seen from the definitions of $M_k, L_k, Q_k$. Property 3. describes a relationship between the context of a character in $L_k$ and the cycle that the character belongs to. According to the definition of cycle, we have

$$\alpha_{Q_k(i)} = L_k[Q_k(i)], L_k[Q_k(Q_k(i))], \ldots, L_k[Q_k^{\alpha_{Q_k(i)}}(i)]$$

from Property 2, we have $M_k[i, 1 : k] = [L_k[Q_k(i)], L_k[Q_k(Q_k(i))], \ldots, L_k[Q_k^{\alpha_{Q_k(i)}}(i)]].$ Comparing $M_k[i, 1 : k]$ and $\alpha_{Q_k(i)}$, this property is immediately true because the cycle $\alpha_{Q_k(i)}$ will repeat itself in $M_k[i, 1 : k]$ at each position $j \in [1, k]$ satisfying $(j - 1) % \alpha_{Q_k(i)} = 0$, where $\%$ is the integer modulo operator.

Therefore once we have all the cycles extracted from $L_k$, retrieving the character that is $l$ character(s) cyclic far away from the character $L_k[i]$ in the original text $S$ in a time complexity of $O(1)$, where $l \in [0, k - 1]$. To achieve this aim, we can resort to the backward BWT algorithm[1] as mentioned in Section II. When we apply the backward BWT in the forward direction starting from $L_k[i]$, we will visit the characters in $\alpha_i$ one by one with a period length of $\alpha_i$. We introduce 2 size-$N$ vectors $X, Y$ to store those cycles. We use the vector $X$ to store all the cycles consecutively in the order that they are discovered when running the original backward BWT against unvisited characters in $L_k$, using the index $i$ to represent the character $L_k[i]$. Then we maintain the relative head and tail positions of each cycle in $Y$. The vector $Y$ contains two different kinds of values: non-negative values and negative values. If $Y[i] \geq 0$, $L_k[X[i + Y[i]]$ is the end character of the cycle; otherwise, $L_k[X[i + Y[i]]$ is the head character of the cycle. The algorithm MakeXY is described in Fig.5, and this requires only $O(N)$ time and space.

After constructing the vector $X, Y$, given $i$ and $l$, we can use a function GetSite to find the character that is $l$ character(s) cyclic far away from the character $L_k[i]$ in the original text $S$ within $O(1)$ time. The detail of the algorithm GetSite is

```plaintext
1: procedure MakeXY($Q_k$)
2: \> $Q_k$ is a vector mapping the position of each character in $F_k$ to its position in $L_k$
3: \> initialize the array visited[1, N]
4: \> $visited[1, N] \leftarrow -1$
5: \> $r, j, v, u \leftarrow 0$
6: \> for $i \leftarrow 1$, len do
7: \> \> if $visited[j] \neq -1$ then
8: \> \> \> for $t \leftarrow 0, u - 1$ do
9: \> \> \> \> $Y[v + t] \leftarrow u - 1 - t$
10: \> \> \> end for
11: \> \> \> $Y[v + u - 1] \leftarrow -(u - 1)$
12: \> \> \> $v \leftarrow v + u$
13: \> \> \> $u \leftarrow 0$
14: \> \> \> for $t \leftarrow r + 1, N$ do
15: \> \> \> \> if $visited[t] \neq -1$ then
16: \> \> \> \> \> $r \leftarrow t$
17: \> \> \> \> \> \> $j \leftarrow r$
18: \> \> \> \> \> \> \> break
19: \> \> \> end if
20: \> \> \end for
21: \> \> end if
22: \> \end for
23: \> if $u > 0$ then
24: \> \> \> for $t \leftarrow 0, u - 1$ do
25: \> \> \> \> $Y[v + t] \leftarrow u - 1 - t$
26: \> \> \> end for
27: \> \> $Y[v + u - 1] \leftarrow -(u - 1)$
28: \> \end if
29: \> return $X, Y$
30: \end procedure
```

Fig. 5. The algorithm for making vector $X$ and $Y$ using $Q_k$.
omitted due to its simplicity.

Given an algorithm to sort the all the suffix of a text $S$ into lexicographical order, we can modify it into a suffix sorting algorithm using $L_k$, by employing the function MakeXY and GetSite. Using such an algorithm as the foundation, and adding necessary components, we come to a unified framework computing both forward and backward ST, as is shown in Fig.6

**B. An algorithm to compute both forward and backward ST**

The $O(n \log k)$ algorithm of Nong and Zhang’s utilizes the doubling technique, but still can not avoid unnecessary scanning even when all the groups have been found. This is a major drawback of Manber-Myers[15] algorithm, which makes it outperformed in practice by an ad hoc string sorting method, optimized for sorting short string [19].

The doubling technique was introduced in [31] and first used for the construction of the suffix array in [15]. Define the $h$-order of the suffixes as their order when sorting lexicographically, considering only the initial $h$ symbols of each suffix. The doubling technique is based on the following property:

**Property 4:** (Manber and Myers).Sorting the suffixes using, for each suffix $S_i$, the position in the $h$-order of $S_i$ as its primary key, and the position of $S_{i+h}$ in the same order as its secondary key, yields the $2h$-order. This property allows each sorting pass double the depth to which the suffixes are sorted, thus only a logarithmic number of passes is needed to sort all the suffixes, and gives an overall worst-case time bound of $O(n \log n)$.

Among those doubling algorithms, the Larsson and Sadakane algorithm(LS algorithm, for short)[19] is the current "leader". It is not only much faster than the suffix tree based methods but also outperform the linear algorithms.

The LS algorithm is based on the observation that usually in suffix sorting, the final positions of most of the suffixes are determined by only the first few symbols of each suffix. Therefore, it utilizes a ternary-split quicksort and a ternary string-sorting strategy to remove unnecessary scanning and idle reorganizing of already sorted suffixes. Thanks to a very clever data organization it only uses $8n$ bytes. Even more surprisingly, the whole algorithm fits in two pages of clean and elegant C code.

Combining the Larsson-Sadakane algorithm and our result obtained in last subsection, we obtain a new algorithm to compute both forward and backward ST, which can have better performance.

1) **The algorithm:** The algorithm utilizes the framework BackwardST described in Fig.6, thus only the part of SuffixSort is given here. Unlike the description above, we omit the procedure of MakeD. Given $Q_k$, the procedure SuffixSort outputs $D$ directly.

The algorithm is shown in Fig.7. The algorithm uses an array $I$ to hold the sorting result, to be more specific, $I[i]$ holds the position of the first character of the suffix in the conceptual cycles array. An auxiliary integer array $V$, is employed to maintain constant-time access to the positions of the suffixes in $I$. Furthermore, we introduce the following definitions:

**Definition 2:** A maximal sequence of adjacent rows in the sorted suffix matrix which have the same initial $h$ symbols is a *group*. A group containing at least two suffixes is an *unsorted group*. A group containing only one suffix is a *sorted group*.

---

**Fig. 6.** An algorithm framework to compute Forward and Backward ST based on suffix sorting

**Fig. 7.** algorithm of SuffixSort employing the idea of LS algorithm
A maximal sequence of adjacent sorted groups is a combined sorted group.

Definition 3: The group number of a group that occupies the subarray $I[f \cdots g]$ is $g$.

During the sorting, the array $V$ stores group numbers. $V[i] = g$ reflects that suffix $i$ is currently in group number $g$. To simplified the description of the algorithm, we introduce another conceptual array $L$ that holds the lengths of unsorted groups and combined sorted groups in positions corresponding to their leftmost elements. To distinguish between these, we store positive numbers for unsorted groups and negative numbers (the negated length) for combined sorted groups. Thus, if the subarray $I[f \cdots g]$ is an unsorted group, we store $g - f + 1$ in $L[f]$; if it is a combined sorted group, we store $-(g - f + 1)$ instead. $L$ can be further superimposed on $I$ during implementation.

The first step of the algorithm places the suffixes - represented as numbers 0 through $N$ - into $I$, sorted according to the first symbol of each suffix. After this step, $I$ is in 1-order. We initialize $V$ and $L$ accordingly. Then a number passes for further sorting follow. At the beginning of the $j$th such pass, $I$ is in $h$-order, where $h = 2^j - 1$. In this pass, we only sort the unsorted group using the group number of suffix $i + h$; here we need to locate the correct suffix using GetSite($i, h$). We then split groups between suffixes with non-equal strings, updating $V$ and $L$. When setting the lengths in $L$, we combine adjacent groups so that they can be efficiently skipped over in subsequent passes. The key to the good performance of this algorithm is the groups lengths stored in $L$, which allow us to skip over sorted groups completely while we continue to process unsorted groups. The subroutine in step 5 using ternary-split quicksort[32], would generate three parts: one with elements smaller than the pivot, one with elements equal to the pivot, and one with larger elements. The smaller and larger parts are then processed recursively while the equal part is left as is, since its elements are already correctly placed.

2) Complexity analysis: As mentioned above, employing the function GetSite only requires $O(1)$ time and $O(N)$ space, as a result the algorithm SuffixSort would have the same $O(N \log k)$ time and $O(N)$ space complexity as LS algorithm.

IV. DISCUSSION

In this paper, we explored some underlying relations between forward and backward ST, showing that any suffix sorting algorithm used in forward ST, can be used in the backward ST. The advantage of our framework is if better suffix sorting algorithm is discovered, better performance of the backward ST can be achieved. However, not all the suffix sorting algorithms can be used efficiently in our framework. For example, if the linear suffix sorting algorithm proposed by Karkkainen and Sanders[20] is applied, the extra $k$ steps access in each cycle, would lead to $k \times N$ extra steps in the worst case, thus the complexity of the algorithm would increase to $O(kN)$.

REFERENCES


