Indexing Integer Strings Using Local Difference Bounds

Daniel Gibney
Department of Computer Science, University of Texas at Dallas, TX, USA

Kaamil Kaka
Department of Computer Science, University of Texas at Dallas, TX, USA

Sharma V. Thankachan
Department of Computer Science, North Carolina State University, TX, USA

Abstract

Time series data can often be represented as a string $T[1..n]$ over an integer alphabet Σ. A standard approach for indexing such strings is the *compressed suffix tree*, which supports efficient exact pattern matching using $\mathcal{O}(n \log |\Sigma|)$ bits of space. However, when $|\Sigma|$ is close to n, as is often the case for time-series data, this yields little to no space savings over the classical $\Theta(n \log n)$-bit suffix tree.

In this work, we study a different parameter that is often much smaller than $|\Sigma|$: the *maximum absolute difference* between consecutive values in T, denoted by Δ. Although representing T in $\mathcal{O}(n \log \Delta)$ bits is straightforward, supporting efficient pattern matching within this space bound remains challenging. By leveraging succinct data structure techniques, particularly the FM-index, we obtain a compressed index occupying $n \log \Delta + \mathcal{O}(n)$ bits. Given a query pattern $P[1 \,..\, m]$, the index answers counting queries in $\mathcal{O}(m \log \Delta)$ time and reporting queries in $\mathcal{O}\big(m \log \Delta + occ \cdot \log n \cdot \log \Delta\big)$ time, where occ denotes the number of occurrences of P in T. In addition, the structure supports suffix array and inverse suffix array queries in $\mathcal{O}(\log n \cdot \log \Delta)$ time.

Our construction is conceptually simple. We show that the local-difference bound induces a strong form of locality in the Burrows–Wheeler Transform (BWT), allowing the transformed text to be decomposed into regions that each use only a small local alphabet of size $\mathcal{O}(\Delta)$. This makes it possible to adapt the classical FM-index so that its space usage depends on Δ rather than the overall alphabet size, while still supporting efficient pattern matching queries.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases String Algorithms, Pattern Matching, Suffix Trees, Compact Data Structures

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.151

Supplementary Material
Software (Source Code): `https://github.com/dangibney/Delta_Bounded_Time_Series_Index` [7]
archived at `swh:1:dir:b33cf856db1f9e72d95c00136bf351670d04bfb2`

Funding *Sharma V. Thankachan*: U.S. NSF awards CCF-2316691 and CCF-2315822.

1 Introduction

In text indexing, we are given a string $T[1 \,..\, n]$, called the text, and we preprocess it to build a data structure, called the index, that enables efficient pattern-matching queries. For an input string $P[1 \,..\, m]$, called a pattern, two basic queries are typically considered: counting queries, which return the number of substrings of T that match P, and reporting queries, which list the starting positions (called occurrences) of these substrings in T.

A central goal is to support these queries using space close to that required to represent T itself. Standard solutions achieve optimal query time but, as we will see, can fall short of optimal space for particular families of strings. We assume that both T and P are over an integer alphabet of size $\sigma = n^{\mathcal{O}(1)}$. The classic suffix tree supports counting queries in $\mathcal{O}(m)$ time and reporting queries in $\mathcal{O}(m + occ)$ time, where occ is the number of occurrences of

P in T [31]. The suffix array is a simpler, array-based alternative that supports the same queries [21]. However, both data structures require $\mathcal{O}(n \log n)$ bits of space. Compressed indexes, such as compressed suffix arrays (CSA) [13] and the FM-index [5], reduce the space usage to $\mathcal{O}(n \log \sigma)$ bits, at the cost of a slight increase in query times. However, when σ is polynomial in n, this bound becomes $\mathcal{O}(n \log n)$ bits, eliminating the advantage over classical suffix trees.

To overcome this limitation, we consider a different parameter that is often much smaller than σ. Specifically, we define $\Delta \geq \max_{1 \leq i < n} |T[i+1] - T[i]|$, which bounds the absolute difference between adjacent characters. This parameter is natural for discretized signals, such as time-series data, where consecutive values change gradually. It is also meaningful in stochastic settings: for a simple symmetric random walk on the integers, we have $\Delta = 1$, yet the maximum value attained over n steps is $\Theta(\sqrt{n})$ in expectation [25]. Thus, even when Δ is constant, the range of values in T can be large.

Observe that such a string T can be represented in $\mathcal{O}(n \log \Delta)$ bits by storing $T[1]$ and the successive differences $T[i+1] - T[i]$. While standard techniques support constant-time random access within this space, building an index that enables efficient pattern matching remains open. To address this, we consider the following problem:

Problem: Differentially Encoded Indexing.
Given $\Delta \geq 2$ and $T[1\,..\,n]$, a length-n string of integers such that $|T[i+1] - T[i]| \leq \Delta$ for all $1 \leq i < n$, we seek a data structure to answer queries of the following form:

Input: A length-m string $P[1\,..\,m]$ of integers (called a *pattern*).
Output: A *reporting query* asks to output the set of all occurrences of P in T, that is $\{i_1, i_2, \ldots, i_{occ}\} \subseteq [1\,..\,n]$ such that $P = T[i_j\,..\,i_j + m)$ for each j. A *counting query* asks to return *occ*, the size of that set.

We restrict our attention to $\Delta \geq 2$, since even if the maximum difference has absolute value 1, a bound of $\Delta \geq 2$ trivially holds. Our main result is summarized in Theorem 1.

▶ **Theorem 1.** *Let $\tau \geq 1$ be a user-chosen sampling parameter. There exists a solution for Differentially Encoded Indexing that occupies*

$$n \log \Delta + \mathcal{O}(n) + \mathcal{O}\left(\frac{n \log n}{\tau}\right)$$

bits of space, supports counting queries in $\mathcal{O}(m \log \Delta)$ time and reporting queries in

$$\mathcal{O}(m \log \Delta + occ \cdot \tau \log \Delta)$$

time, where occ denotes the number of occurrences of P in T. In addition, the index supports suffix array and inverse suffix array queries in $\mathcal{O}(\tau \log \Delta)$ time.

By setting $\tau = \log n$, we obtain the bounds stated in the abstract.

Our approach is based on a simple observation about the classical FM-index. Given the Δ bound on differences between adjacent characters, the LF-mapping (described in Section 2) is constrained to a limited range. This locality allows us to decompose the Burrows–Wheeler Transform (BWT) of the text into blocks, each defined over a local alphabet of size $\mathcal{O}(\Delta)$. These blocks can be transformed and concatenated into a sequence over an alphabet of size $\mathcal{O}(\Delta)$, which can then be stored efficiently using a wavelet tree [12].

1.1 Related Work and Applications

Many recent breakthroughs in text indexing have focused on repetition-aware indexing [2, 6, 16, 23, 24]. The goal is to design indexes whose space depends on the text's substring complexity δ, defined as $\delta = \max_k d_k(T)/k$, where $d_k(T)$ denotes the number of distinct substrings of length k in T [19, 27]. The measure δ lower-bounds several classical measures of compressibility, such as the size of the LZ77 parse and the number of runs in the Burrows–Wheeler Transform (BWT) [15, 17].

It is known that $\mathcal{O}(\delta \log(n/\delta))$ words of space (i.e., $\mathcal{O}(\delta \log(n/\delta) \cdot \log n)$ bits) are necessary to store the text, and indexes matching this bound can support reporting queries in $\mathcal{O}(m + (1 + occ) \log^{\epsilon} n)$ time [18], where $\epsilon > 0$ is an arbitrarily small constant fixed at construction time. A recent result also supports suffix array and inverse suffix array queries in $\mathcal{O}(\log^{4+\epsilon} n)$ time within this space bound [16].

While δ can be significantly smaller than n for highly repetitive strings, it is not always the right measure. In particular, one can construct strings for which $n \log \Delta$ is asymptotically smaller than both $\delta \log(n/\delta) \log n$ and $n \log \sigma$. Thus, our index is best viewed as complementary, especially in settings where repetition-aware indexes are not space-efficient.

An alternative and closely related perspective on our construction is through the lens of *wavelet forests* [1, 9], in which a sequence is partitioned into multiple subsequences that are each represented using separate wavelet trees. Conceptually, our construction can be interpreted as a structured wavelet-forest decomposition in which the partition is induced by ranges of the F column of the BWT. The key additional ingredient is the use of carefully chosen offsets, which ensure that every block is represented over a local alphabet of size $\mathcal{O}(\Delta)$.

The theoretical presentation in this paper uses a single wavelet tree over the concatenation of the shifted block sequences. However, our implementation instead stores the blocks using multiple wavelet trees. This practical choice can significantly reduce space usage further in practice, since many blocks induce substantially smaller local alphabets than the worst-case bound of 3Δ. We discuss this implementation choice further in Section 4.

In time-series analysis, most work on pattern matching focuses on approximate matching under distance measures such as dynamic time warping. A prominent example is the UCR suite of Rakthanmanon et al. [26]. For exact matching, however, these methods require at least a linear scan of T, i.e., $\mathcal{O}(n)$ time, making them less efficient than our approach in the worst case. The iSAX method of Shieh and Keogh [29] partially mitigates this by organizing the data into a hierarchical structure. However, queries still require scanning the portions of T associated with the leaves, leading again to linear-time behavior. Similar hierarchical approaches based on discrete Fourier coefficients [20] exhibit the same limitation. We refer to [4] for a broader survey of time-series indexing techniques.

2 Preliminaries

We work in the word-RAM model with word size $\Theta(\log n)$. We assume $\Delta = o(n)$, since otherwise no space savings over classical suffix trees or suffix arrays is possible.

Rank, Select, and Count. Let $B[1\,..\,n]$ be a binary array. We define:

$$\mathsf{RANK}(i, B) = \sum_{k=1}^{i} B[k], \quad \mathsf{SELECT}(i, B) = \min\{j \mid \mathsf{RANK}(j, B) = i\}.$$

That is, $\mathsf{RANK}(i, B)$ counts the number of 1's in $B[1\,..\,i]$, and $\mathsf{SELECT}(i, B)$ returns the position of the i-th 1 in B. There exist data structures using $\mathcal{O}(n)$ bits of space that support both operations in $\mathcal{O}(1)$ time and can be constructed in $\mathcal{O}(n)$ time [3, 14].

For a general string $S[1\,..\,n]$ over alphabet Σ, we extend RANK as

$$\mathsf{RANK}_c(i, S) = |\{k \le i \mid S[k] = c\}|.$$

These queries can be supported using *wavelet trees* [12, 22], which occupy $n \log \sigma + o(n \log \sigma)$ bits of space and answer rank queries in $\mathcal{O}(\log \sigma)$ time. By internally using the bit map structures of Golynski et al. [10] this space can be further reduced to $n \log \sigma + o(n)$ bits.

Wavelet trees also support select queries. For a character c and integer $k \ge 1$, we define

$$\mathsf{SELECT}_c(k, S) = \min\{i \mid \mathsf{RANK}_c(i, S) = k\},$$

that is, the position of the k-th occurrence of c in S. These queries can be supported in $\mathcal{O}(\log \sigma)$ time using the same space.

Furthermore, wavelet trees support range-count queries of the form

$$\mathsf{COUNT}(a, b, i, j) = |\{k \in [i\,..\,j] \mid S[k] \in [a\,..\,b)\}|$$

in $\mathcal{O}(\log \sigma)$ time. For example, $\mathsf{COUNT}(a, b, 1, |S|)$ is the number of characters in the entire sequence whose values lie in $[a\,..\,b)$, which can be computed in $\mathcal{O}(\log \sigma)$ time [1].

Strings and Suffix Arrays. Let $T = T[1]T[2]\ldots T[n]$ be a string over an integer alphabet $\Sigma = [0\ldots\sigma)$. For $1 \le i \le j \le n$, we denote by $T[i\,..\,j]$ the substring from position i to j, and by $T[i\,..\,j)$ the substring $T[i\,..\,j-1]$.

The suffix array $\mathrm{SA}[1\,..\,n]$ lists the starting positions of all suffixes of T in lexicographic order. That is, $\mathrm{SA}[j] = i$ if and only if $T[i\,..\,n]$ is the j-th smallest suffix. The inverse suffix array $\mathrm{ISA}[1\,..\,n]$ satisfies $\mathrm{ISA}[i] = j$ if $\mathrm{SA}[j] = i$.

For a pattern P, its suffix range is the interval $[sp(P), ep(P)]$ such that all suffixes prefixed by P appear in $\mathrm{SA}[sp(P)\,..\,ep(P)]$. Therefore, the set $\{\mathrm{SA}[k] \mid k \in [sp(P), ep(P)]\}$ gives all occurrences of P in T. Note that the suffix range is empty if P does not occur in T.

Burrows–Wheeler Transform and FM-index. The Burrows–Wheeler Transform (BWT) of $T\$$, denoted $\mathrm{BWT}[1\,..\,n+1]$, is defined as follows. Let $\mathrm{SA}[1\,..\,n+1]$ be the suffix array of $T\$$, where $\$$ is a special character appended to T that is lexicographically smaller than all other characters. Then

$$\mathrm{BWT}[i] = \begin{cases} T[\mathrm{SA}[i]-1] & \text{if } \mathrm{SA}[i] > 1, \\ \$ & \text{if } \mathrm{SA}[i] = 1. \end{cases}$$

Let $C[c]$ denote the number of characters in T that are strictly smaller than c. The FM-index supports pattern matching via the *LF-mapping*, defined as

$$\mathsf{LF}(i) = C[\mathrm{BWT}[i]] + \mathsf{RANK}_{\mathrm{BWT}[i]}(i, \mathrm{BWT}).$$

The LF-mapping maps a position corresponding to the suffix of $T\$$ beginning at position $\mathrm{SA}[i]$ to the position corresponding to the suffix beginning at position $\mathrm{SA}[i]-1$, effectively prepending one character.

[1] Large-alphabet sequence representations can support exact-symbol rank queries in $\mathcal{O}(\log\log\sigma)$ time within comparable space bounds [11]. However, our construction also requires access and ordered range-count queries of the form $\mathsf{COUNT}(a, b, i, j)$. These operations are directly supported by wavelet trees.

Using this property, the FM-index performs *backward search*. Starting from the full suffix range $[1, n+1]$, we process the pattern P from right to left. Given the current range $[sp, ep]$ for suffixes prefixed by $P[i+1\,..\,m]$, we update it for $P[i\,..\,m]$ as:

$$sp = C[P[i]] + \mathsf{RANK}_{P[i]}(sp-1, \mathrm{BWT}) + 1, \quad ep = C[P[i]] + \mathsf{RANK}_{P[i]}(ep, \mathrm{BWT}).$$

After processing all characters, the resulting interval gives the suffix range of P. Finally, we define LF^{-1} as the inverse permutation of LF. Suffix array and inverse suffix array queries are also supported by FM-index through the LF-mapping and sampled suffix array/inverse suffix array. We detail this procedure, along with the slight modification needed to access the current character, in Sections 3.2 and 3.3.

3 Our Solution

The primary data structure, which supports LF-mapping, consists of a bit vector supporting rank and select queries together with a wavelet tree constructed over a transformed version of the BWT. To accomplish reporting, suffix array, and inverse suffix array queries, we additionally store a structure consisting of a bit vector and an array of sampled SA / ISA values.

3.1 The Data Structure

We first calculate the minimum character $a_{\min}$ of T and subtract it from all entries. Next, we append to T the character \$, which is the lexicographically smallest character and does not occur elsewhere in T. Consider the BWT of T\$. We group the indices of $\mathrm{BWT}[1\,..\,n+1]$ into blocks $B_0, B_1, B_2, \ldots$, where $B_j = [s_j, s_{j+1})$ is the maximal range of indices such that $F[i] \in [j\Delta, (j+1)\Delta)$ for all $i \in B_j$. We consider $T[n]$ the first entry of B_0 since it is cyclically followed by the \$ symbol.

Note that block indices are numbered starting from 0, while all strings and arrays are indexed starting from 1. We indicate the first entry of each block in a bit vector $\mathsf{BLOCK\_BV}$ of length $n+1$, where

$$\mathsf{BLOCK\_BV}[i] = 1 \quad \text{if and only if } i = s_j \text{ for some } j.$$

This structure uses $\mathcal{O}(n)$ bits and supports rank and select queries in $\mathcal{O}(1)$ time.

Let $\mathrm{BWT}[B_j] = \mathrm{BWT}[s_j\,..\,s_{j+1})$. Our first observation is that the Δ bound on differences in T implies a form of locality for the corresponding BWT values. The only *exceptional* positions are the indices $i \in \{1, \mathsf{LF}^{-1}(1)\}$, corresponding to the suffixes beginning at the boundaries of T. At these positions, the adjacency relation between $\mathrm{BWT}[i]$ and $F[i]$ does not correspond to adjacent characters in T, since they involve the sentinel \$. In particular, $F[1] = \$$ and $\mathrm{BWT}[1] = T[n]$, while $F[\mathsf{LF}^{-1}(1)] = T[1]$ and $\mathrm{BWT}[\mathsf{LF}^{-1}(1)] = \$$.

We formalize this in the following lemma.

▶ **Lemma 2.** *For any block B_j and for all $i \in B_j$, if $i \notin \{1, \mathsf{LF}^{-1}(1)\}$, then*

$$\mathrm{BWT}[i] \in [(j-1)\Delta, (j+2)\Delta).$$

Proof. For such i, the values $\mathrm{BWT}[i]$ and $F[i]$ correspond to adjacent characters in T, and hence satisfy

$$|\mathrm{BWT}[i] - F[i]| \le \Delta.$$

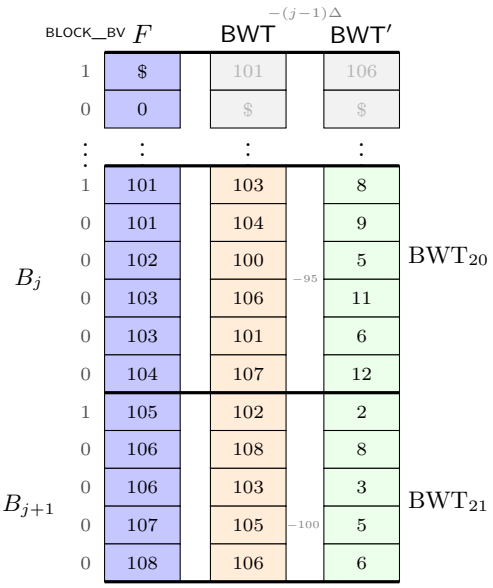

Figure 1 Block-structured view of F, BWT, and BWT′ for $\Delta = 5$. Grayed-out entries correspond to the exceptional sentinel-related positions.

Since $F[i] \in [j\Delta, (j+1)\Delta)$, we have

$$\mathrm{BWT}[i] \le F[i] + \Delta < (j+1)\Delta + \Delta = (j+2)\Delta,$$

and

$$\mathrm{BWT}[i] \ge F[i] - \Delta \ge j\Delta - \Delta = (j-1)\Delta.$$ ◀

Based on the lemma above, we now construct the rest of our data structure:

(Wavelet Tree) For each block B_j, we define a shifted sequence BWT_j by offsetting each value by $(j-1)\Delta$, i.e., for $i \in B_j$,

$$\mathrm{BWT}_j[i - s_j + 1] = \mathrm{BWT}[i] - (j-1)\Delta.$$

See Figure 1. By Lemma 2, for all $i \in B_j \setminus \{1, \mathrm{LF}^{-1}(1)\}$, we have

$$\mathrm{BWT}[i] \in [(j-1)\Delta, (j+2)\Delta).$$

Therefore,

$$\mathrm{BWT}_j[i - s_j + 1] \in [0, 3\Delta).$$

We treat the two exceptional entries corresponding to \$ and $T[n]$ separately by assigning them the value 3Δ in their corresponding shifted blocks (for the purposes of wavelet tree alphabet size) and store their original values explicitly.
We now concatenate all shifted blocks to obtain

$$\mathrm{BWT}' = \mathrm{BWT}_0\,\mathrm{BWT}_1 \cdots \mathrm{BWT}_t\,.$$

Since each shifted block lies over an alphabet of size at most $3\Delta + 1$, the entire sequence BWT′ also lies over an alphabet of size $\mathcal{O}(\Delta)$. We therefore store BWT′ in a single wavelet tree occupying

$$(n+1)\log(3\Delta+1) + o(n) = n\log\Delta + \mathcal{O}(n)$$

bits while supporting rank, select, and range-count queries in $\mathcal{O}(\log\Delta)$ time.

- **(Suffix Array Samples)** Let τ be the given sampling parameter. We sample text positions $1, 1+\tau, 1+2\tau, \ldots$. For each sampled text position p, we compute its inverse suffix array value $\mathrm{ISA}[p]$ and mark that row in a bit vector $\mathsf{B_{SA}}$ of length $n+1$, i.e., $\mathsf{B_{SA}}[\mathrm{ISA}[p]] = 1$. We also store the sampled text positions in an array SA', ordered by their corresponding inverse suffix array values. Equivalently, if $p_1, p_2, \ldots, p_t$ are the sampled text positions sorted so that

  $$\mathrm{ISA}[p_1] < \mathrm{ISA}[p_2] < \cdots < \mathrm{ISA}[p_t],$$

  then $\mathrm{SA}'[k] = p_k$ for $1 \le k \le t$. The number of samples is at most $\mathcal{O}\left(\frac{n}{\tau}\right)$, so the total space for suffix array samples is $\mathcal{O}\left(\frac{n \log n}{\tau}\right)$ bits.
- Finally, we store $T[n]$ and the smallest character $a_{\min}$ using $\mathcal{O}(\log n)$ bits.

3.2 Counting and Reporting Queries

We now turn to detailing the querying procedure. To this end, the following lemma bounds how many adjacent blocks a given character in BWT of T can appear in.

▶ **Lemma 3.** *For two distinct blocks B_j and $B_{j'}$ with $j' \le j-3$, we have*

$$\mathrm{BWT}[B_j] \cap \mathrm{BWT}[B_{j'}] \subseteq \{\$, T[n]\}.$$

Proof. Let $i \in B_j$ and $i' \in B_{j'}$, and suppose $i, i' \notin \{1, \mathrm{LF}^{-1}(1)\}$. Then

$$|\mathrm{BWT}[i] - F[i]| \le \Delta, \qquad |\mathrm{BWT}[i'] - F[i']| \le \Delta.$$

Since $F[i] \in [j\Delta, (j+1)\Delta)$ and $F[i'] \in [j'\Delta, (j'+1)\Delta)$, and since $j' \le j-3$, we have

$$F[i] \ge j\Delta \qquad \text{and} \qquad F[i'] < (j'+1)\Delta \le (j-2)\Delta.$$

Thus

$$F[i] - F[i'] > 2\Delta.$$

Therefore

$$\mathrm{BWT}[i] - \mathrm{BWT}[i'] \ge (F[i] - \Delta) - (F[i'] + \Delta) = F[i] - F[i'] - 2\Delta > 0.$$

Hence $\mathrm{BWT}[i] \ne \mathrm{BWT}[i']$. The only excluded rows are the two exceptional rows, whose BWT values are \$ and $T[n]$. ◀

Backward Search. To initialize the backward search, we first initialize the suffix range $[sp_{m+1}, ep_{m+1}] = [1, n+1]$. To perform a backward step from the range of $P[k+1\,..\,m]$ to that of $P[k\,..\,m]$, suppose we are given the current range $[sp_{k+1}, ep_{k+1}]$. Then we compute

$$sp_k = C[P[k]] + \mathsf{RANK}_{P[k]}(sp_{k+1} - 1, \mathrm{BWT}) + 1,$$

and

$$ep_k = C[P[k]] + \mathsf{RANK}_{P[k]}(ep_{k+1}, \mathrm{BWT}).$$

Thus, it suffices to support the operations $C[c]$ and $\mathsf{RANK}_c(\mathsf{pos}, \mathrm{BWT})$ efficiently.

Computing $\mathsf{RANK}_c(\mathsf{pos}, \mathbf{BWT})$. Let $q = \left\lfloor \frac{c}{\Delta} \right\rfloor$. By Lemma 2, any non-exceptional occurrence of c in BWT can only lie in the three blocks

$$B_{q-1},\ B_q,\ B_{q+1}.$$

Let

$$s_t = \mathsf{SELECT}(t+1, \mathsf{BLOCK\_BV})$$

denote the start of block B_t, and let s_{t+1} denote the start of the next block, or $n+2$ if B_t is the last block. For each candidate block B_t, define

$$r_t(\mathsf{pos}) = \min(\mathsf{pos}, s_{t+1} - 1).$$

Then

$$\mathsf{RANK}_c(\mathsf{pos}, \mathrm{BWT}) = \sum_{t=q-1}^{q+1} \mathsf{COUNT}\Big(c - o_t,\ c - o_t + 1,\ s_t,\ r_t(\mathsf{pos})\Big) + \chi,$$

where $o_t = (t-1)\Delta$, terms corresponding to nonexistent blocks or empty position ranges are treated as zero, and

$$\chi = \begin{cases} 1 & \text{if } c = T[n] \\ 0 & \text{otherwise.} \end{cases}$$

The additional term accounts for the exceptional occurrence of $T[n]$ at the first position of the BWT.

This rank computation also allows us to determine whether $c \in T$. In particular,

$$c \in T \iff \mathsf{RANK}_c(n+1, \mathrm{BWT}) > 0.$$

If $c \notin T$, then any pattern P containing c has zero occurrences in T.

Computing $C[c]$. Recall that $C[c]$ denotes the number of characters in T that are strictly smaller than c. Since every character appears in BWT with the same multiplicity as in T, we may equivalently compute these counts using BWT.

Let $q = \left\lfloor \frac{c}{\Delta} \right\rfloor$. All characters strictly smaller than $q\Delta$ occur before block B_q in the F column. Since rows are indexed starting from 1, the number of such rows is $s_q - 1$, where

$$s_q = \mathsf{SELECT}(q+1, \mathsf{BLOCK\_BV}).$$

It remains to count characters in the range $[q\Delta, c)$. By Lemma 2, any non-exceptional occurrence of a character in this range can only occur in the three blocks

$$B_{q-1},\ B_q,\ B_{q+1}.$$

Therefore,

$$C[c] = (s_q - 1) + \sum_{t=q-1}^{q+1} \mathsf{COUNT}\Big(q\Delta - o_t,\ c - o_t,\ s_t,\ s_{t+1} - 1\Big) + \chi_C,$$

where $o_t = (t-1)\Delta$, terms corresponding to nonexistent blocks are treated as zero, and

$$\chi_C = \begin{cases} 1 & \text{if } q\Delta \le T[n] < c \\ 0 & \text{otherwise.} \end{cases}$$

The correction term accounts for the exceptional occurrence of $T[n]$ at the first position of the BWT.

Reporting. To report occurrences, we locate the suffix-array value corresponding to each row $i \in [sp_1 \,..\, ep_1]$. Starting from the row i, we repeatedly apply LF-mapping until a sampled suffix-array row is reached, that is, until

$$\mathsf{B_{SA}}[\mathrm{LF}^k(i)] = 1$$

for some $k \geq 0$.

During this traversal, suppose the current row is j. We determine the block B_t that contains j, retrieve the corresponding shifted value from BWT', and recover the original character by adding the block offset:

$$\mathrm{BWT}[j] = \mathrm{BWT}'[j] + (t-1)\Delta.$$

Since the shifted blocks were concatenated in their original order, position j in BWT' corresponds directly to position j in BWT. The recovered character is then used to evaluate the LF-mapping.

If $\mathrm{LF}^k(i)$ is the first sampled row reached during this traversal, then

$$\mathrm{SA}[i] = \mathrm{SA}'\big[\mathsf{RANK}(\mathrm{LF}^k(i), \mathsf{B_{SA}})\big] + k.$$

This follows from the identity

$$\mathrm{LF}(\mathrm{ISA}[p]) = \mathrm{ISA}[p-1].$$

At most $(\tau - 1)$ LF-steps are required before reaching a sampled row. Since each LF-step takes $\mathcal{O}(\log \Delta)$ time, the total reporting time per occurrence is $\mathcal{O}(\tau \log \Delta)$.

3.3 Suffix and Inverse Suffix Array Queries

Suffix array queries can be answered in $\mathcal{O}(\tau \log \Delta)$ time using the same LF-mapping procedure as in reporting. Starting from a row j, we repeatedly apply LF until a sampled row is reached. Then, we recover $\mathrm{SA}[j]$ from the stored sample and add the number of LF-mapping applications.

To support inverse suffix array queries within the same time bound, we additionally store, for every sampled text position $p \in \{1, \tau+1, 2\tau+1, \dots\}$, the corresponding row $\mathrm{ISA}[p]$ in an array indexed by sampled text positions. Let

$$i' = 1 + \tau \left\lceil \frac{i-1}{\tau} \right\rceil$$

be the smallest sampled text position such that $i' \geq i$. We retrieve the sampled row $\mathrm{ISA}[i']$ directly from this array and then apply LF exactly $i' - i$ times. Since $\mathrm{LF}(\mathrm{ISA}[p]) = \mathrm{ISA}[p-1]$ for every $p > 1$, it follows inductively that $\mathrm{LF}^k(\mathrm{ISA}[p]) = \mathrm{ISA}[p-k]$. Therefore, $\mathrm{ISA}[i] = \mathrm{LF}^{i'-i}(\mathrm{ISA}[i'])$.

Since at most $\tau - 1$ LF-steps are performed and each LF-step requires $\mathcal{O}(\log \Delta)$ time, inverse suffix array queries are supported in $\mathcal{O}(\tau \log \Delta)$ time. The additional sampled array occupies $\mathcal{O}\left(\frac{n \log n}{\tau}\right)$ bits, maintaining the same asymptotic space complexity.

4 Preliminary Experiments

We conducted preliminary experiments to compare our index against an integer wavelet tree FM-index implementation based on the SDSL library [8]. Our goal was to evaluate the practical impact of parameterizing the index by the local-difference bound Δ, particularly for sequences whose alphabet size grows with the length of the text despite having small adjacent differences.

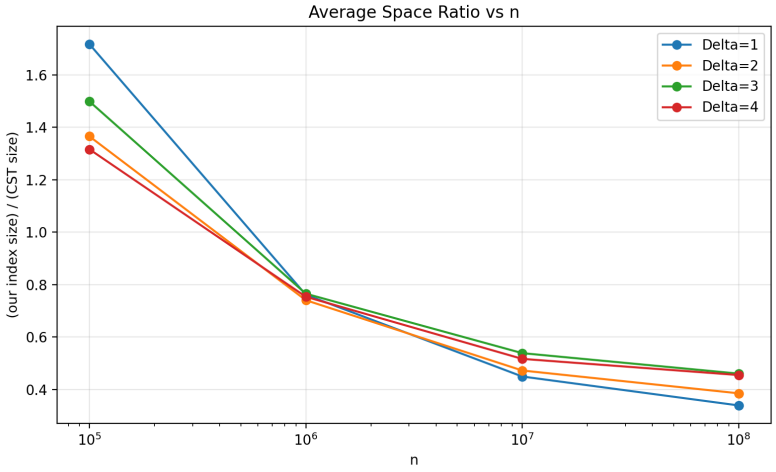

Figure 2 Average ratio between the size of our index and the size of the SDSL FM-index over 10 independently generated random walks for each parameter pair (n, Δ). Values below 1 indicate that our structure uses less space.

Dataset generation. To generate synthetic datasets, we used bounded random walks over the integers. Starting from the value 0, each subsequent value was obtained by adding a uniformly chosen integer from $[-\Delta, \Delta]$. Thus, every generated sequence satisfies $|T[i+1] - T[i]| \leq \Delta$. We generated datasets for $\Delta \in \{1, 2, 3, 4\}$ and for sequence lengths ranging from 10^5 to 10^8.

For each pair (n, Δ), we generated 10 independent random walks. Query patterns were sampled directly from the generated text to ensure that occurrences existed in the dataset. We tested pattern length $m = 100$, sampling 100 patterns for each pattern length from every generated walk.

Practical Implementation. While our theoretical presentation uses a single wavelet tree over the concatenation of all shifted blocks, our implementation instead stores the blocks using multiple wavelet trees. This design choice is motivated by practical considerations and supported by our experimental evaluation. Although every block is guaranteed to use an alphabet of size at most $\mathcal{O}(\Delta)$, many blocks induce substantially smaller local alphabets in practice. Storing blocks independently therefore allows the implementation to exploit these smaller effective alphabets, often reducing space usage further. Furthermore, it is not clear how such a BWT decomposition could be performed without the Δ bound, while maintaining efficient LF-mapping.

Our implementation additionally performs a rank-compression step on the input alphabet prior to index construction. To support this mapping efficiently, we store the distinct characters using a succinct bit-vector representation with rank and select support. The implementation itself is built using the SDSL library and uses integer wavelet trees to store transformed BWT blocks. The implementation used in our experiments is publicly available at `https://github.com/dangibney/Delta_Bounded_Time_Series_Index`.

Compared indexes. We compared our structure against an FM-index implementation based on SDSL's compressed suffix array with wavelet trees. Both structures included suffix array samples. Since SDSL operates over compact alphabets, the input integers were first rank-compressed prior to index construction. For both indexes, we measured index size and counting query time.

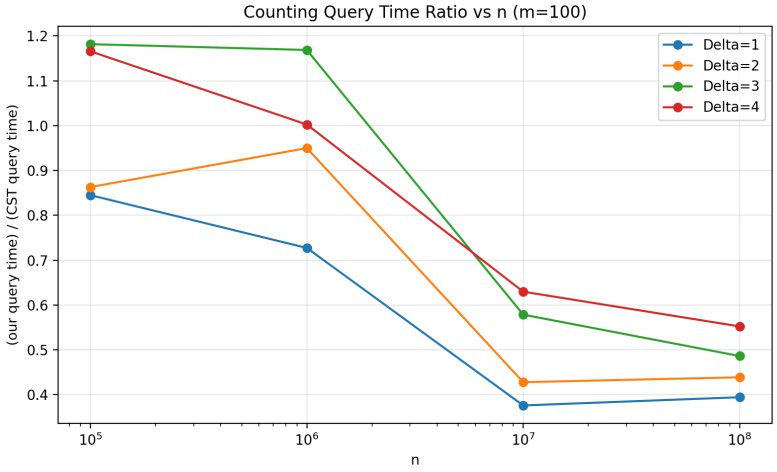

Figure 3 Average ratio between counting-query running times of our index and the SDSL FM-index for patterns of length $m = 100$, averaged over 10 independently generated random walks for each parameter pair (n, Δ). Values below 1 indicate that our structure is faster.

Experimental environment. All experiments were conducted on a machine running Ubuntu 24.04 LTS equipped with an Intel Core i5-1235U processor and 64 GiB of RAM.

Preliminary results. Figure 2 reports the average ratio between the size of our index and the size of the SDSL FM-index across the 10 random walks generated for each parameter pair (n, Δ). For larger inputs, our structure consistently used less space than the baseline implementation. The improvement became more pronounced for smaller values of Δ, which is consistent with the theoretical dependence of our structure on $\log \Delta$ rather than the effective alphabet size.

Figure 3 reports the average ratio between counting-query running times for patterns of length $m = 100$, again averaged over the 10 generated random walks for each parameter pair. A ratio below 1 indicates that our structure outperformed the SDSL baseline. For larger datasets, our structure achieved competitive and often faster query times, particularly for smaller values of Δ.

Overall, these preliminary experiments suggest that local-difference-aware indexing may provide substantial practical space savings while maintaining competitive query performance on data exhibiting small local fluctuations.

5 Discussion

In this work, we introduced a simple compressed indexing framework parameterized by the maximum adjacent difference

$$\Delta \geq \max_{1 \leq i < n} |T[i+1] - T[i]|.$$

Our results show that strong worst-case guarantees can be obtained whenever consecutive values in the text vary slowly, even if the overall alphabet size is large. We also note that as a consequence of a result by Sadakane, by utilizing an additional $\mathcal{O}(n)$ bits, the remaining functionality of a suffix tree can be supported with polylogarithmic query time [28].

A natural direction for future work is to move beyond worst-case local variation and instead parameterize the index by the *average* adjacent difference. For example, one could define

$$\overline{\Delta} = \frac{1}{n-1} \sum_{i=1}^{n-1} |T[i+1] - T[i]|.$$

In many real-world time-series datasets, large jumps may occur only rarely, while most adjacent values differ by a very small amount. In such settings, the maximum difference Δ may be dominated by a small number of outliers and therefore fail to accurately capture the effective local complexity of the sequence.

This raises the question of whether one can design compressed indexes whose space depends primarily on $\overline{\Delta}$, or more generally on the distribution of adjacent differences, while still supporting efficient pattern matching queries. One possible direction is to allow blocks with variable local alphabet sizes, adapting to the observed fluctuations in different regions of the text. Another possibility is to combine our approach with entropy-compressed or repetition-aware indexing techniques in order to simultaneously exploit local smoothness and global repetitiveness.

More broadly, our work suggests that structural properties arising from local numerical constraints may provide an alternative foundation for compressed indexing beyond traditional measures such as alphabet size or substring repetitiveness.

Finally, recent work has demonstrated that inverse suffix array queries can be supported in doubly logarithmic time within succinct space [30]. A natural question is whether these techniques can be adapted to our setting.

References

1 Eric Chiu and Dominik Kempa. Wavelet forests revisited, 2026. doi:10.48550/arXiv.2604.11338.

2 Anders Roy Christiansen, Mikko Berggren Ettienne, Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Optimal-time dictionary-compressed indexes. *ACM Trans. Algorithms*, 17(1):8:1–8:39, 2021. doi:10.1145/3426473.

3 David Richard Clark. *Compact PAT Trees*. Ph.D. thesis, University of Waterloo, Waterloo, Ontario, Canada, 1996. doi:10.5555/287799.

4 Hui Ding, Goce Trajcevski, Peter Scheuermann, Xiaoyue Wang, and Eamonn J. Keogh. Querying and mining of time series data: experimental comparison of representations and distance measures. *Proc. VLDB Endow.*, 1(2):1542–1552, 2008. doi:10.14778/1454159.1454226.

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

6 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020. doi:10.1145/3375890.

7 Daniel Gibney, Kaamil Kaka, and Sharma V. Thankachan. dangibney/Delta_Bounded_Time_Series_Index. Software, swhId: swh:1:dir:b33cf856db1f9e72d95c00136bf351670d04bfb2 (visited on 2026-08-11). URL: https://github.com/dangibney/Delta_Bounded_Time_Series_Index, doi:10.4230/artifacts.27610.

8 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014. doi:10.1007/978-3-319-07959-2_28.

9 Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J. Puglisi. Fixed block compression boosting in fm-indexes: Theory and practice. *Algorithmica*, 81(4):1370–1391, 2019. doi:10.1007/S00453-018-0475-9.

10 Alexander Golynski, Roberto Grossi, Ankur Gupta, Rajeev Raman, and S. Srinivasa Rao. On the size of succinct indices. In Lars Arge, Michael Hoffmann, and Emo Welzl, editors, *Algorithms - ESA 2007, 15th Annual European Symposium, Eilat, Israel, October 8-10, 2007, Proceedings*, Lecture Notes in Computer Science, pages 371–382. Springer, 2007. doi:10.1007/978-3-540-75520-3_34.

11 Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 368–373. ACM Press, 2006. URL: http://dl.acm.org/citation.cfm?id=1109557.1109599.

12 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*, pages 841–850. ACM/SIAM, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644250.

13 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.

14 Guy Joseph Jacobson. *Succinct static data structures*. Carnegie Mellon University, 1988.

15 Dominik Kempa and Tomasz Kociumaka. Resolution of the burrows-wheeler transform conjecture. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1002–1013. IEEE, 2020. doi:10.1109/FOCS46700.2020.00097.

16 Dominik Kempa and Tomasz Kociumaka. Collapsing the hierarchy of compressed data structures: Suffix arrays in optimal compressed space. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, pages 1877–1886. IEEE, 2023. doi:10.1109/FOCS57990.2023.00114.

17 Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In Ilias Diakonikolas, David Kempe, and Monika Henzinger, editors, *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 827–840. ACM, 2018. doi:10.1145/3188745.3188814.

18 Tomasz Kociumaka, Gonzalo Navarro, and Francisco Olivares. Near-optimal search time in δ-optimal space, and vice versa. *Algorithmica*, 86(4):1031–1056, 2024. doi:10.1007/S00453-023-01186-0.

19 Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Toward a definitive compressibility measure for repetitive sequences. *IEEE Trans. Inf. Theory*, 69(4):2074–2092, 2023. doi:10.1109/TIT.2022.3224382.

20 Xiao-Ying Liu and Chuan-Lun Ren. Fast subsequence matching under time warping in time-series databases. In *International Conference on Machine Learning and Cybernetics, ICMLC 2013, Tianjin, China, July 14-17, 2013*, pages 1584–1590. IEEE, 2013. doi:10.1109/ICMLC.2013.6890855.

21 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.

22 Gonzalo Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014. doi:10.1016/J.JDA.2013.07.004.

23 Gonzalo Navarro. Indexing highly repetitive string collections, part I: repetitiveness measures. *ACM Comput. Surv.*, 54(2):29:1–29:31, 2022. doi:10.1145/3434399.

24 Gonzalo Navarro. Indexing highly repetitive string collections, part II: compressed indexes. *ACM Comput. Surv.*, 54(2):26:1–26:32, 2022. doi:10.1145/3432999.

25 José Luis Palacios. On the simple symmetric random walk and its maximal function. *The American Statistician*, 62(2):138–140, 2008.

26 Thanawin Rakthanmanon, Bilson J. L. Campana, Abdullah Mueen, Gustavo E. A. P. A. Batista, M. Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn J. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In Qiang Yang, Deepak Agarwal, and Jian Pei, editors, *The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012*, pages 262–270. ACM, 2012. doi:10.1145/2339530.2339576.

27 Sofya Raskhodnikova, Dana Ron, Ronitt Rubinfeld, and Adam D. Smith. Sublinear algorithms for approximating string compressibility. *Algorithmica*, 65(3):685–709, 2013. doi:10.1007/S00453-012-9618-6.

28 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/S00224-006-1198-X.

29 Jin Shieh and Eamonn J. Keogh. *i*sax: indexing and mining terabyte sized time series. In Ying Li, Bing Liu, and Sunita Sarawagi, editors, *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, August 24-27, 2008*, pages 623–631. ACM, 2008. doi:10.1145/1401890.1401966.

30 Sharma V. Thankachan. Compressed inverse suffix arrays. *CoRR*, abs/2607.17287, 2026. doi:10.48550/ARXIV.2607.17287.

31 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.