The SBC-Tree: An Index for Run-Length Compressed Sequences *†

Mohamed Y. Eltabakh[1] Wing-Kai Hon[2] Rahul Shah[1]
Walid G. Aref[1] Jeffrey S. Vitter[1]
[1]Department of Computer Science, Purdue University
[2]Department of Computer Science, National Tsing Hua University
[1]{meltabak, rahul, aref, jsv}@cs.purdue.edu, [2]wkhon@cs.nthu.edu.tw

ABSTRACT

Run-Length-Encoding (RLE) is a data compression technique that is used in various applications, e.g., time series, biological sequences, and multimedia databases. One of the main challenges is how to operate on (e.g., index, search, and retrieve) compressed data without decompressing it. In this paper, we introduce the *S*tring *B*-tree for *C*ompressed sequences, termed the *SBC-tree*, for indexing and searching RLE-compressed sequences of arbitrary length. The SBC-tree is a two-level index structure based on the well-known String B-tree and a 3-sided range query structure [7]. The SBC-tree supports pattern matching queries such as *substring matching*, *prefix matching*, and *range search* operations over RLE-compressed sequences. The SBC-tree has an optimal external-memory space complexity of $O(N/B)$ pages, where N is the total length of the compressed sequences, and B is the disk page size. *Substring matching*, *prefix matching*, and *range search* execute in an optimal $O(\log_B N + \frac{|p|+T}{B})$ I/O operations, where $|p|$ is the length of the compressed query pattern and T is the query output size. The SBC-tree is also dynamic and supports *insert* and *delete* operations efficiently. The insertion and deletion of all suffixes of a compressed sequence of length m take $O(m \log_B(N+m))$ amortized I/O operations. The SBC-tree index is realized inside PostgreSQL. Performance results illustrate that using the SBC-tree to index RLE-compressed sequences achieves up to an order of magnitude reduction in storage, while retains the optimal search performance achieved by the String B-tree over the uncompressed sequences.

1. INTRODUCTION

Current databases store massive amounts of data, especially in text and sequence formats, e.g., time series databases, biological sequences, medical record, and digital libraries. With such massive amounts of data, data compression techniques, e.g., [14, 23, 34, 42, 48, 49], gain significant importance to achieve compact data representation. Compressing the data is proven to improve the system performance, e.g., [41]. It reduces significantly the size of the data, the number of I/O operations, and the buffer requirements. One of the main challenges is how to operate on (e.g., index, search, and retrieve) compressed data without decompressing it. Some compression techniques complicate significantly the representation of the data, and hence make efficient searching over the compressed data almost impossible. Other compression techniques, e.g., Burrows-Wheeler Transform (BWT), and Run-Length Encoding (RLE), allow direct searching over the compressed data.

Run-Length-Encoding (RLE) [23] is a compression technique that replaces the consecutive repeats of an element x by one occurrence of x along with x's frequency, i.e., the repeat length. For example, a sequence $S = AAAAEEEBBBBBBB$ has an RLE-compressed form $S' = A4E3B7$. RLE is used to compress data from various domains, e.g., time series, biological, and multimedia databases. Several in-memory algorithms have been proposed to search RLE compressed sequences, e.g., [1, 2, 3, 5, 13, 22]. However, none of the proposed algorithms address the problem of indexing and searching compressed data using external memory techniques [46]. RLE is also used inside the database management system C-Store [41] to compress sorted columns that have few distinct values. C-Store allows some database operators to execute directly over the RLE compressed data, e.g., aggregate operators. However, performing more complex operations, e.g., indexing and substring searching RLE compressed sequences, has not been addressed yet.

In this paper, we propose the SBC-tree (String B-tree for Compressed sequences) for indexing and searching RLE-compressed sequences of arbitrary length. The SBC-tree is a two-level index structure as illustrated in Figure 1. The first level is a modified version of the String B-tree proposed in [18], and the second level is the optimum 3-sided range query structure proposed in [7]. The 3-sided structure is built on top of the leaf entries of the modified String B-tree. The SBC-tree supports *substring*, *prefix*, and *range search*

*Rahul Shah and Jeffrey S. Vitter acknowledge the support of the National Science Foundation under grant number CCF–0621457.
†Walid G. Aref acknowledges the support of the National Science Foundation under grant number IIS-0093116.

EDBT'08, March 25–30, 2008, Nantes, France.

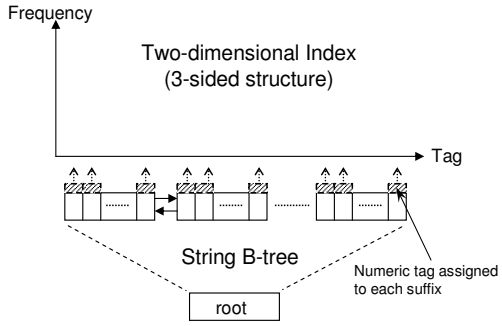


Figure 1: The SBC-tree structure.

operations over RLE-compressed sequences.

We formalize our problem as follows. Let S be an RLE-compressed sequence of length n in the form $S =$ '$x_1f_1\ x_2f_2\ ...\ x_nf_n$', x_j is a character in the alphabet Σ, and $f_j \geq 1$ is the frequency of x_j. We call x_jf_j an *RLE-character*. Sequence S has n *RLE-suffixes*, i.e., RLE-Suffixes(S) $= \{x_jf_j\ x_{j+1}f_{j+1}\ ...\ x_nf_n \mid 1 \leq j \leq n\}$. The length of the decompressed sequence of S is the sum of the character frequencies forming S. That is, $|decompressed(S)| = \sum_{j=1}^{n} f_j$. The decompressed sequence of S has $\sum_{j=1}^{n} f_j$ suffixes. The n RLE-suffixes of S are a subset of the total $\sum_{j=1}^{n} f_j$ suffixes. The remaining $\sum_{j=1}^{n} f_j - n$ suffixes are called *implicit-suffixes*, as they are not stored explicitly among the RLE-suffixes. Given a set of K RLE-compressed sequences $\Delta = \{S_1, S_2, ..., S_K\}$, the proposed SBC-tree achieves the following: (1) store the sequences in their compressed form, (2) index only the RLE-suffixes of the RLE-compressed sequences, i.e., index n RLE-suffixes instead of $\sum_{j=1}^{n} f_j$ suffixes for each sequence, and (3) efficiently answer substring matching queries over the stored sequences.

The SBC-tree has an optimal external-memory space complexity of O(N/B) pages, where N is the total length of the compressed sequences and B is the disk page size. The insertion and deletion of all suffixes of a compressed sequence of length m take O($m \log_B(N+m)$) amortized, and worst-case I/O operations, respectively. *Substring matching*, *prefix matching*, and *range search* execute in an optimal O($\log_B N + \frac{|p|+T}{B}$) I/O operations, where $|p|$ is the length of the RLE-compressed query pattern and T is the query output size.

In order to put the SBC-tree into practice and facilitate its implementation inside current database management systems, we present a variant of the SBC-tree that uses the R-tree instead of the 3-sided structure. This variant has no provable worst-case theoretical bounds for search operations. However, it is more practical from a systems implementation point of view and also has good empirical results.

The contributions of this paper are summarized as follows:

1. We introduce the SBC-tree index for indexing and substring searching RLE-compressed sequences of arbitrary lengths. The SBC-tree is realized inside PostgreSQL.
2. The SBC-tree has provable worst-case theoretical bounds for the external-memory space requirements and search operations. The SBC-tree is the first compressed index structure that is dynamic and operates optimally in external memory with respect to the size of the compressed data.
3. The experimental results illustrate that using the SBC-tree to index RLE-compressed sequences achieves up to an order of magnitude reduction in storage, up to 30% reduction in I/Os for the insertion operations, and retains the optimal search performance achieved by the String B-tree over the uncompressed sequences.
4. The SBC-tree supports complex search operations, e.g., *regular expression searching*. The experimental results illustrate that the SBC-tree achieves up to 80% reduction in I/Os for *regular expression searching* compared to the String B-tree.

The rest of the paper is organized as follows. In Section 2, we discuss the related work. In Section 3, we present the component substructures that make the SBC-tree. We present the SBC-tree structure along with its update and search algorithms in Sections 4 and 5. The experimental results are presented in Sections 6. Section 7 contains concluding remarks.

2. RELATED WORK

The concept of searching compressed data is introduced in [4, 44]. Several in-memory algorithms have been proposed to search various formats of compressed data. Algorithms for searching RLE-compressed sequences include substring matching [2, 3, 44], approximate pattern matching [30], edit distance [6, 13], and longest common subsequence [5, 22]. However, processing RLE-compressed sequences in external memory has not been addressed yet. The proposed SBC-tree addresses the challenge of indexing and searching RLE-compressed sequences in external memory.

Algorithms over other compression schemes include searching data compressed in Lempel-Ziv (LZ) [1, 36], antidictionaries [40], and Burrows-Wheeler Transform (BWT) [12]. For applications such as entropy compressed text, the encoding scheme is complex, and hence the search mechanisms have to be carefully engineered. For this purpose, several in-memory pattern matching data structures which compress the text to high-order entropy have been proposed. These structures are based on Burrows Wheeler Transform(BWT) [19] and Compressed Suffix Arrays(CSA) [24, 25]. However, indexing and searching compressed data in external memory is more challenging, and no external memory structures analogous to the structures above exist. In fact, recent studies show that no data structures that achieve high-order entropy can be externalized and still achieve O(polylog N + T/B) I/O term in the query bound [15]. Moreover, these data structures cannot be effectively dynamic (support insertions and deletions) [29, 37]. Compared to these schemes, the proposed SBC-tree is simple, dynamic, and achieves optimal search performance. While RLE may not be widely used as BWT or LZ, there are many data sets that can be effectively compressed using RLE.

Indexing compressed sequences is closely tied to text and sequence indexing. A model for sequence databases, called *SEQ*, has been proposed in [39]. SEQ models different types of sequence data and defines a set of operators to query the sequences. Several well-known index structures for text indexing have been proposed. These structures include suffix trees [26, 32, 47], suffix binary search trees [28], suffix arrays [20, 26, 31], inverted files [38], tries [21, 35], B-trees [8, 16], and the prefix B-tree [9]. Several variants of these structures have been proposed to efficiently index strings of unbounded length. For example, the persistent suffix trees have been proposed in [11, 27]. A buffer management strategy for a practical construction of suffix trees has been proposed in [43]. The String B-tree which is an external memory structure for suffix arrays in the form of a B-tree is proposed in [18].

Using existing text indexing structures to index RLE-compressed sequences is not straightforward because these structures and their search mechanisms are based on storing all suffixes of the underlying sequences. The challenge is how to efficiently answer pattern matching queries, e.g., *substring matching*, *prefix matching*, and *range search*, while indexing only a small subset of the suffixes.

3. SBC-TREE COMPONENT STRUCTURES

In this section, we present the data structures that we use to construct the SBC-tree. In Section 3.1, we describe the String B-tree that is the basis for the first level of the SBC-tree, and in Section 3.2, we describe the 3-sided structure that is the basis for the second level of the SBC-tree.

3.1 The String B-tree

The String B-tree [18] is a data structure for indexing strings of arbitrary length, where index nodes store the strings' logical keys instead of the strings themselves. A string logical key is the start position of the string on disk. Suffixes of a given string have different logical keys depending on their start positions in the string. The logical keys are sorted inside the String B-tree according to the lexicographic order of the corresponding suffixes (See Figure 2).

The String B-tree is a combination of the B-tree [16] and the Patricia trie [35], where the entries inside each B-tree node are organized in a Patricia trie structure instead of a sequential array. We illustrate in Figure 2 the String B-tree for a set of strings. The positions of the strings on disk are presented in Figure 2(a). The leaf entries of the String B-tree contain the logical keys of all suffixes ordered in lexicographic order from left to right. The right-most key in each node propagates to the parent node (Figure 2(b)). The node highlighted in Figure 2(c) contains a Patricia trie for substrings, "te", "tend", "tent", "tenuate", "tl", and "tlas". Each Patricia trie node stores the position at which the substrings under the node's subtree first differ along with the branching characters. For example, the first position at which the strings illustrated in Figure 2(c) differ is position 1, and the branching characters are e and l.

Searching the String B-tree is done by performing two root-to-leaf path traversals to locate the first and last keys satisfying the query. All the keys between the first and last keys are the query answer.

The String B-tree has good performance and worst-case theoretical bounds in answering pattern matching queries. The following lemma states the theoretical bounds of the String B-tree [18].

Lemma 1. ([18] Theorem 2.2):

a) The space complexity of the String B-tree is $O(N/B)$ pages, where N is the total length of the strings, and B is the disk page size.

b) The insertion and deletion of all suffixes of a string of length m take $O(m \log_B(N+m))$ I/O operations.

c) A root-to-leaf path traversal to locate the first or last occurrence of pattern p executes in $O(\log_B N + \frac{|p|}{B})$ I/O operations, where $|p|$ is the length of p.

d) Substring searching for pattern p executes in $O(\log_B N + \frac{|p|+T}{B})$ I/O operations, where $|p|$ is the length of p, and T is the query output size.

3.2 The 3-sided Range Query Structure

Given a set of N points in a two-dimensional space, a 3-sided range query is defined as a query with three parameters ($a1$, $a2$, $b1$), where $a1$ and $a2$ specify the lower and upper limits over the first dimension, respectively, and $b1$ specifies the lower limit over the second dimension. The answer to the query is all points (x, y), where $a1 \leq x \leq a2$ and $y \geq b1$. (See Figure 3).

The 3-sided range query structure [7] is an external memory structure that is based on the external memory priority search tree [33] and the persistent B-tree [10, 45]. The 3-sided structure consists of a "base-tree" and a set of substructures. Each node in the base-tree holds a set of $O(B^2)$ points in a substructure, termed *B^2-sized structure*, that occupies $O(B)$ disk pages. A point is stored in at most one *B^2-sized structure*, but it can be replicated more than once in that structure. For a particular *B^2-sized structure*, if *occ* points qualify for a given query, then (occ/B) I/O operations are performed over that structure to report the points. The height of the base-tree in a 3-sided structure is $O(\log N / \log(B^2)) = O(\log_B N)$, where N is the total number of points.

The 3-sided range query structure has an optimal worst-case theoretical bound for the update and 3-sided range query operations. The following lemma states the theoretical bounds of the 3-sided structure [7].

Lemma 2. ([7] Theorem 6):

a) The space complexity of the 3-sided range query structure is $O(N/B)$ pages, where N is the number of points in the space, and B is the disk page size.

b) The insertion and deletion of a point take $O(\log_B N)$ worst-case I/O operations.

c) The 3-sided range query executes in $O(\log_B N + \frac{T}{B})$ worst-case I/O operations, where T is the output size.

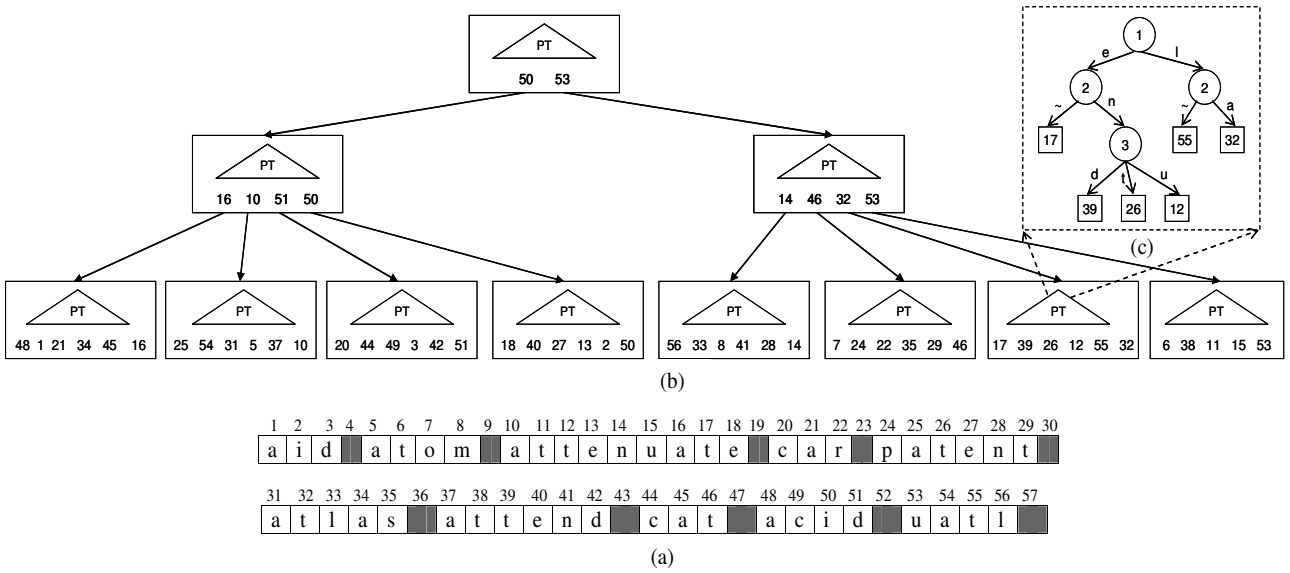


Figure 2: (a) Strings on disk, (b) The String B-tree for all suffixes, (c) The Patricia trie inside one node.

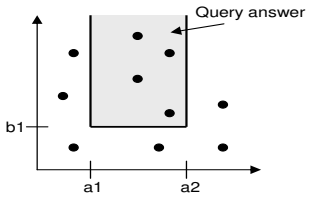


Figure 3: 3-sided query ($a1$, $a2$, $b1$).

4. SBC-TREE DESIGN AND STRUCTURE

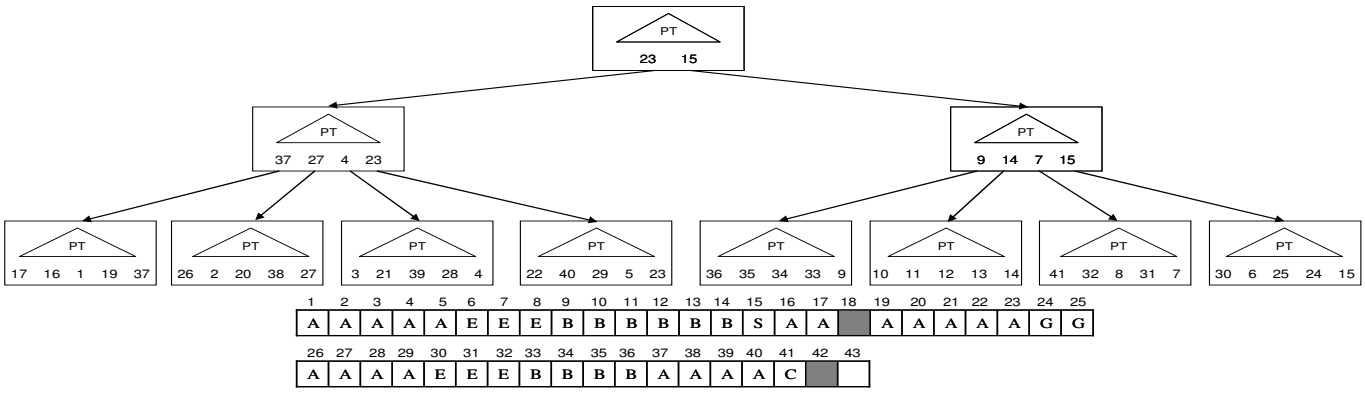
Indexing the RLE-suffixes of RLE-compressed sequences means that the generated index will not contain all suffixes of the original (decompressed) sequence. Therefore, the String B-tree cannot be used directly to search the compressed sequences. The structure and search mechanism of the String B-tree (See Section 3.1) are based on storing all sequences' suffixes inside the index. The following example demonstrates the problem.

Example 1. Assume we are indexing two sequences, $S_1 = A5E3B6S1A2$ and $S_2 = A5G2A4E3B4A4C1$. We present the RLE-suffixes of S_1 and S_2 in Figure 4(b). The *order* column represents the lexicographic order of the suffixes without compression. The number of the uncompressed and RLE- suffixes of S_1 and S_2 is 40 and 12 suffixes, respectively. In Figures 4(a) and 4(c), we illustrate the String B-tree of the uncompressed and RLE- suffixes, respectively, assuming a maximum B-tree node size of five entries. Consider a *substring match* searching for pattern $p = AAEEEBBBB$ over the uncompressed index (Figures 4(a)). The search will return two hits with the suffixes starting at positions 28 and 4 on the disk. However, applying the same query over the RLE-suffixes (Figure 4(c)), where p is compressed to $A2E3B4$, will not return any hits. The reason is that the suffixes starting with $A2E3B4$ are not stored in the index. Instead, they are implicit-suffixes and are included in longer RLE-suffixes, i.e., the RLE-suffix $A5E3B6S1A2$ of S_1 and $A4E3B4A4C1$ of S_2 implicitly contain the string $A2E3B4$.

The trick to answer the *substring matching* query correctly over the RLE-suffixes is to take the implicit-suffixes into account while searching the compressed index. This is done by mapping the query pattern $p = A2E3B4$ into $p' = A2^+E3B4$, where $A2^+$ means repeats of letter A of length larger than or equal to 2. As a result, RLE-suffixes whose prefix explicitly matches p or include implicit-suffixes whose prefix matches p will be an answer to the query. For example, the RLE-suffixes $A5E3B6S1A2$ and $A4E3B4A4C1$ starting at positions 1 and 16 on the disk (Figure 4(c)) are an answer to the query above. The RLE-suffix $A5E3B6S1A2$ includes the implicit-suffix $A2E3B6S1A2$ whose prefix matches p, and the RLE-suffix $A4E3B4A4C1$ includes the implicit-suffix $A2E3B4A4C1$ whose prefix matches p. The following rule formalizes the *substring matching* query pattern mapping.

Rule 1. A *substring matching* query pattern $p = x_1f_1\ x_2f_2\ ...\ x_nf_n$ over RLE-suffixes is mapped into pattern $p' = x_1f_1^+\ x_2f_2\ ...\ x_nf_n$, where $x_1f_1^+$ means repeats of character x_1 of length larger than or equal to f_1.

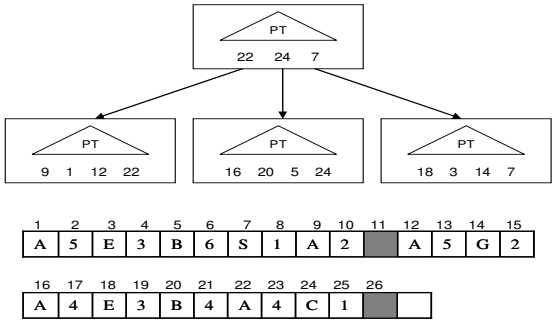
Although the query pattern mapping returns the correct answer to *substring matching* queries, the mapping results in another problem. The RLE-suffixes that satisfy the mapped query pattern are not guaranteed to be contiguous inside the String B-tree index. Hence, the String B-tree search mechanism that assumes the answer set to be contiguous in the index tree is no longer feasible. If $p' = x_1f_1^+\ x_2f_2\ ...\ x_nf_n$ is the mapped query pattern, then between any two RLE-suffixes starting with $x_1(f_1+i)\ x_2f_2\ ...\ x_nf_n$ and $x_1(f_1+i+1)\ x_2f_2\ ...\ x_nf_n$, where i ≥ 0, there can be an unbounded number of RLE-suffixes that do not satisfy the query. That is, incrementing the frequency of x_1 causes the answer set not to be contiguous. For example, the



(a) The String B-tree for the uncompressed suffixes

RLE-suffixes	order
A5 E3 B6 S1 A2	2
E3 B6 S1 A2	10
B6 S1 A2	7
S1 A2	12
A2	1
A5 G2 A4 E3 B4 A4 C1	3
G2 A4 E3 B4 A4 C1	11
A4 E3 B4 A4 C1	5
E3 B4 A4 C1	9
B4 A4 C1	6
A4 C1	4
C1	8

(b) The RLE-suffixes



(c) The String B-tree for the RLE-suffixes

Modified RLE-suffixes	order
Ψ1 A5 E3 B6 S1 A2	1
A1 E3 B6 S1 A2	6
E1 B6 S1 A2	12
B1 S1 A2	9
S1 A2	14
A1	3
Ψ1 A5 G2 A4 E3 B4 A4 C1	2
A1 G2 A4 E3 B4 A4 C1	7
G1 A4 E3 B4 A4 C1	13
A1 E3 B4 A4 C1	5
E1 B4 A4 C1	11
B1 A4 C1	8
A1 C1	4
C1	10

(d) The SBC-tree indexed RLE-suffixes

Figure 4: Indexing the uncompressed and RLE- suffixes of sequences $A5E3B6S1A2$ and $A5G2A4E3B4A4C1$.

two RLE-suffixes $A5E3B6S1A2$ and $A4E3B4A4C1$ starting at positions 1 and 16, respectively, satisfy the query pattern $p' = A2^{+}E3B4$ (See Figure 4(c)). However, the two RLE-suffixes in-between, i.e., $A5G2A4E3B4A4C1$ and $A4C1$, which start at positions 12 and 22, respectively, do not satisfy the query. The proposed SBC-tree index provides a solution to this problem.

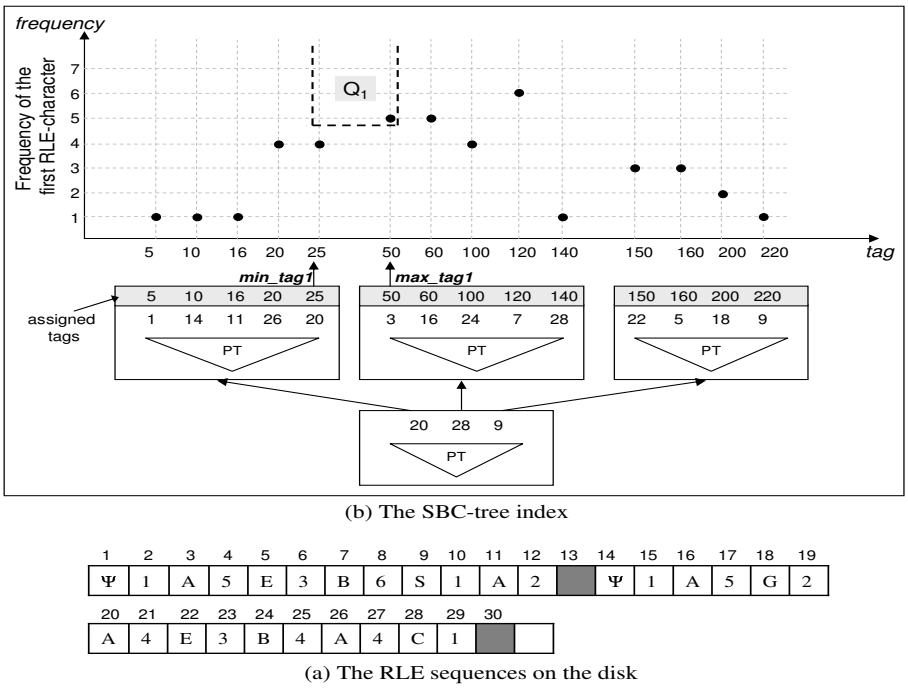
4.1 The SBC-tree Structure

The SBC-tree is a two-level index structure. The first level is a modified version of the String B-tree, and the second level is the 3-sided index structure proposed in [7] (Refer to Figure 5). The first level of the SBC-tree indexes modified versions of the RLE-suffixes where the frequency of the first RLE-character in each RLE-suffix is set to 1. For example, instead of indexing the RLE-suffixes in Figure 4(b), the SBC-tree indexes the modified RLE-suffixes in Figure 4(d). First, each sequence in the database is prefixed by a special RLE-character $\Psi 1$, where Ψ is smaller than any character in the alphabet. Then, for each RLE-suffix, we set the frequency of the suffix's first RLE-character to 1. The second level of the SBC-tree indexes points in a two-dimensional space that represent a reference to the RLE-suffix (the X-axis) and the exact frequency of the suffix's first RLE-character (the Y-axis).

An RLE-compressed sequence $S = \Psi 1 \ \ x_1 f_1 \ \ x_2 f_2 \ \ ... \ \ x_n f_n$ is indexed in the SBC-tree as follows:

1. Insert S into the String B-tree as the first RLE-suffix.
2. For $1 \leq i \leq n$, insert suffix $x_i 1 \ \ x_{i+1} f_{i+1} \ \ ... \ \ x_n f_n$ into the String B-tree.
3. Assign a numeric tag to each inserted RLE-suffix (leaf entry) that reflects the entry's position in the index (See Figure 5). Tags from the left-most leaf entry to the right-most leaf entry are of increasing order. Tags are assigned dynamically at the insertion time using an order-maintenance technique [17]. We discuss the assignment of the tags in detail in Section 5.1.
4. The suffix's tag and the frequency of the suffix's first RLE-character are inserted as a point in the 3-sided structure.

In Figure 5, we illustrate the structure of the SBC-tree for the sequences presented in Figure 4(c). Notice that the modified suffixes that are indexed by the String B-tree do not exist on the disk because we set the frequency of the first RLE-character to 1. For example, the second entry in Figure 4(d) with modified suffix $A1E3B6S1A2$ corresponds to the actual suffix $A5E3B6S1A2$ that is stored on the disk at position 1. As a result, we slightly modified the String B-tree insert and search algorithms as follow. An inserted modified RLE-suffix $x_i 1 \ \ x_{i+1} f_{i+1} \ \ ... \ \ x_n f_n$ points to the disk position of the corresponding actual RLE-suffix $x_i f_i \ \ x_{i+1} f_{i+1} \ \ ... \ \ x_n f_n$. At search time, the frequency of the first RLE-character of the retrieved suffix is set to 1 before performing any comparison operation over that suffix.



(b) The SBC-tree index

(a) The RLE sequences on the disk

Figure 5: The SBC-tree for sequences $S_1 = A5E3B6S1A2$ and $S_2 = A5G2A4E3B4A4C1$.

4.2 Answering Substring Matching Queries

Query Definition: Given a query pattern p, where $p = x_1f_1\ x_2f_2\ ...\ x_nf_n$, find all substrings in the database whose prefix matches p.

A *substring matching* query is answered as follows.

1. Map the query pattern p into $p'' = x_1 1\ x_2f_2\ ...\ x_nf_n$.
2. Search the SBC-tree first level, i.e., the String B-tree, for p''. The answer from the String B-tree is a contiguous range specified by two tags, *min_tag* and *max_tag*. *min_tag* and *max_tag* correspond to the first and last RLE-suffixes (in lexicographic order) whose prefixes match p'', respectively.
3. Apply a two-dimensional range query over the SBC-tree second level, where the *tag* dimension ranges from *min_tag* to *max_tag*, and the *frequency* dimension is larger than or equal to f_1. The answer to the range query is the answer to the *substring matching* query.

In Step 1, we map p into $p'' = x_1 1\ x_2f_2\ ...\ x_nf_n$ instead of $p' = x_1f_1^+\ x_2f_2\ ...\ x_nf_n$ because searching for pattern P'' is guaranteed to return a contiguous range in the String B-tree that is specified by *min_tag* and *max_tag*. Therefore, we need only two root-to-leaf paths over the String B-tree to determine these tags (Step 2). In Step 3, we retrieve from the specified range the RLE-suffixes whose first RLE-character has frequency $\geq f_1$.

In Figure 5, we give an example of *substring match* searching for pattern $p = A5E3B4$. The corresponding p' and p'' will be $A5^+E3B4$ and $A1E3B4$, respectively. The search for p'' over the String B-tree returns the two tags *min_tag1*= 25 and *max_tag1*= 50. The corresponding range query, denoted by Q_1, over the 3-sided structure retrieves only one RLE-suffix starting at position 3 on the disk.

The following lemma states the theoretical bound of *substring matching*.

Lemma 3. A *Substring matching* query over an SBC-tree index executes in an optimal $O(\log_B N + \frac{|p|+T}{B})$ I/O operations, where B is the disk page size, N is the total length of the RLE-compressed sequences (also, the number of points in the 2D space), $|p|$ is the length of a RLE-compressed query pattern, and T is the query output size.

PROOF. Lemma 3 can be easily derived from Lemmas 1 and 2, where a root-to-leaf path traversal over the String B-tree executes in $O(\log_B N + \frac{|p|}{B})$ I/O operations (Lemma 1c), and a range query over the 3-sided structure executes in $O(\log_B N + \frac{T}{B})$ I/O operations (Lemma 2c). □

4.3 Answering Prefix Matching Queries

Query Definition: Given a query pattern p, where $p = x_1f_1\ x_2f_2\ ...\ x_nf_n$, find all database sequences whose prefix matches p.

In *prefix matching*, suffixes that satisfy the query have to be prefixes to their sequences, i.e., the suffix is the entire sequence. In this case, implicit-suffixes cannot be an answer to the query because implicit-suffixes are not prefixes to their sequences. Therefore, in *prefix matching*, we do not need to apply the mapping rule (Rule 1) to pattern p.

To answer a *prefix matching* query, we prefix the query pat-

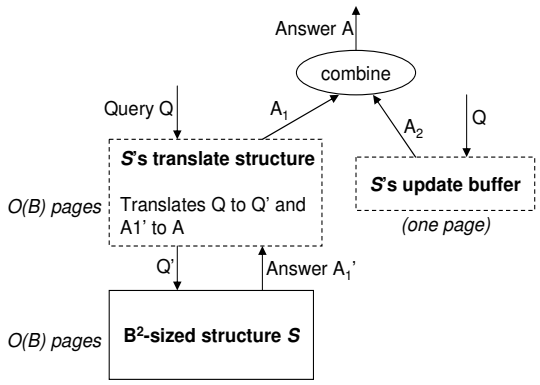


Figure 6: Query processing over the 3-sided structure

tern by $\Psi 1$, and then search the String B-tree. The answer from the String B-tree is a contiguous range that represents the answer set.

4.4 Answering Range Search Queries

Query Definition: Given two query patterns p_1 and p_2, where $p_1 = x_1 f_{x1}\ x_2 f_{x2}\ ...\ x_n f_{xn}$, $p_2 = y_1 f_{y1}\ y_2 f_{y2}\ ...\ y_m f_{ym}$, and p_1 is lexicographically less than p_2, find all database sequences between p_1 and p_2 in lexicographic order.

Range search queries execute in a similar way to *prefix matching* queries. Patterns p_1 and p_2 are prefixed by $\Psi 1$, and then the String B-tree is searched to specify the first pattern larger than or equal to p_1 and the last pattern smaller than or equal to p_2. All patterns in-between belong to the answer set.

The following lemma states the theoretical bound of the *prefix matching* and *range search* queries.

Lemma 4. Prefix matching, and *range search* queries over the SBC-tree index execute in an optimal $O(\log_B N + \frac{|p|+T}{B})$ I/O operations.

The theoretical bounds for the *prefix matching* and *range search* queries are optimal under the assumption that indexing all suffixes is required to answer the *substring matching* queries. Otherwise, a better theoretical bound for *prefix matching* and *range search* queries of $O(\log_B K + \frac{|p|+T}{B})$ I/O operations can be achieved, where K is the number of sequences [18].

5. DESIGN ISSUES

5.1 Updating the SBC-tree

Each leaf entry in the first level of the SBC-tree is assigned a numeric tag that represents the entry's relative position in the tree. The only invariant that we need to maintain for the tags is that tags from the left-most leaf entry to the right-most leaf entry are of increasing order. When a new leaf l is inserted between two leaves l_1 and l_2, l is assigned a tag that is between the tags of l_1 and l_2, i.e., $tag(l_1) < tag(l) < tag(l_2)$. The tag assignment problem arises when the tags of l_1 and l_2 are consecutive, i.e., no tag can be generated between $tag(l_1)$ and $tag(l_2)$. In this case, we need to re-assign the tags to the leaf entries in the vicinity of l to make room for $tag(l)$. Entries that are re-assigned new tags will be deleted from the SBC-tree's second level and are re-inserted with the new tag values.

We first consider the case when no re-labeling is needed. An RLE-suffix is inserted into the String B-tree and is assigned a tag. Then, a point corresponding to that suffix is inserted into the appropriate B^2-*sized structure* inside the 3-sided structure. The inserts over the B^2-*sized structure* are handled in an amortized sense, as proposed in [7], by using *update buffers* (Refer to Figure 6). Each B^2-*sized structure* is assigned a buffer of size one page that holds up to B insertions. When the buffer is full, the B^2-*sized structure* is re-constructed to absorb the items in the buffer. Thus, the B^2-*sized structure* is re-constructed only after B insertions. In [7], it is shown how to re-construct the B^2-*sized structure* in $O(B)$ I/Os. Thus, the amortized cost per insertion is $O(1)$ I/Os. Notice that during the search, when a query hits a particular B^2-*sized structure*, the corresponding *update buffer* is also searched and the results from the two structures are combined (See Figure 6).

Considering the case when a re-labeling is needed, Dietz and Sleator [17] propose an algorithm that maintains dynamically the increasing property of N tags in an amortized $O(\log_2 N)$ CPU time per insertion. That is, on average, each insertion may require re-assigning tags to $\log_2 N$ entries. The updated tags are in a contiguous region. Thus, the $\log_2 N$ tags can be updated in the String B-tree in $O((\log_2 N)/B) = O(\log_B N)$ I/O operations. These tags need to be updated in the B^2-*sized structure*(s) inside the 3-sided structure. These update operations are tricky because a point in a given B^2-*sized structure* can be replicated more than once and we need to update all copies of these points. Therefore, updating the B^2-*sized structure*(s) directly cannot achieve the claimed theoretical bound for the update operations.

To overcome this problem, we maintain a "translate" structure along with each B^2-*sized structure* (See Figure 6). Using this translate structure, we never need to update (delete and re-insert) points in the B^2-*sized structure*. The translate structure consists of $O(B)$ pages with a copy of each point in the corresponding B^2-*sized structure*. The translate structure maintains a mapping between the old and new tags of each point. When a point's tag changes, we only update the new tag of that point in the translate structure. Thus, points in the B^2-*sized structure* are never relabeled. The translate structure holds the points contiguously in the X-order (tag-order), thus $\hat{t}$ points in the translate structure can be relabeled in $O(\hat{t}/B)$ I/Os (using a 2-level B-tree for the translate structure). The points in the "update buffer" do not need to be go through the translate structure. They are always kept up-to-date.

While processing a query, the tags returned from the String B-tree are mapped from their new values to the old values, and then the range query is executed over the B^2-*sized structure*. The tags of the returned points are then mapped from their old values to the new values. The mapping of the tag values can be efficiently performed assuming that $M > B^2$,

where M is the memory size.

Using the translate structures, an insert operation over the SBC-tree that may result in re-labeling $\log_2 N$ points amortized can be executed in order $O(\log_B N + (\log_2 N)/B) = O(log_B N)$ I/Os amortized cost.

The following lemma states the theoretical bounds of the update operations over the SBC-tree.

Lemma 5. The insertion and deletion operations over the SBC-tree execute in $O(m \log_B(N+m))$ amortized, and worst-case I/O operations, respectively, where m is the length of the RLE-compressed sequence.

PROOF. The insertion operation requires: (1) inserting m suffixes into the String B-tree which requires $O(m \log_B(N+m))$ I/O operations (Lemma 1b), (2) possible tag re-labeling in the 3-sided structure which requires $O(m \log_B(N+m))$ amortized I/O operations, and (3) inserting m points into the 3-sided structure which requires $O(m \log_B(N+m))$ I/O operations (Lemma 2b). Therefore, an insertion operation over the SBC-tree requires $O(m \log_B(N+m))$ amortized I/O operations.

The deletion operation requires: (1) deleting m suffixes from the String B-tree which executes in $O(m \log_B(N+m))$ I/O operations (Lemma 1b), and (2) deleting m points from the 3-sided structure which executes in $O(m \log_B(N+m))$ I/O operations (Lemma 2b). Therefore, a deletion operation over the SBC-tree requires $O(m \log_B(N+m))$ worst-case I/O operations. □

5.2 SBC-tree Space Requirements

The SBC-tree structure consists of a String B-tree and a 3-sided structure. The space complexity of the String B-tree is $O(N/B)$ pages (Lemma 1a), and the space complexity of the 3-sided structure is $O(N/B)$ pages (Lemma 2a). Notice that the use of a translate structure of $O(B)$ pages with each B^2*-sized structure* does not change the space complexity stated in Lemma 2a. Based on these bounds, we derive the following lemma.

Lemma 6. The SBC-tree has an optimal external-memory space complexity of $O(N/B)$ pages.

Based on Lemmas 3, 4, 5, and 6, the following theorem states the SBC-tree theoretical bounds.

Theorem. The SBC-tree has an optimal external-memory space complexity of $O(N/B)$ pages. The insertion and deletion of m RLE-suffixes of a compressed sequence execute in $O(m \log_B(N+m))$ amortized and worst-case I/O operations, respectively. The *substring matching*, *prefix matching*, and *range search* operations over the SBC-tree index execute in an optimal $O(\log_B N + \frac{|p|+T}{B})$ I/O operations.

5.3 A Note on Implementation

5.3.1 The Use of R-tree

Although the 3-sided structure is efficient in answering range queries, it is not supported by current database management systems. Our implementation of the 3-sided structure is outside the database engine, i.e., the index data is stored in flat files. In order to put the SBC-tree into practice, we implemented the SBC-tree inside PostgreSQL using the R-tree instead of the 3-sided structure. The search algorithm over the R-tree is the same as that over the 3-sided structure. The SBC-tree using the R-tree has no provable theoretical bounds, but performs well in practice.

5.3.2 The One-level SBC-tree

The structure of the SBC-tree can be simplified, at the expense of the search performance, by dropping the SBC-trees second level, i.e., the two-dimensional index structure. In the one-level SBC-tree, instead of storing the preceding RLE-character of each RLE-suffix in a two-dimensional index, we store the preceding RLE-character inside the RLE-suffix's entry in the String B-tree in place of the tag entries. This simplification improves the space requirements and insertion performance because we do not maintain a second level structure. However, the search performance of the one-level SBC-tree is not as efficient as the search performance of the two-level SBC-tree. The reason is that the search, e.g., *substring matching*, *prefix matching*, or *range search*, over the one-level SBC-tree is performed by scanning the keys in the range specified by the two tags, *min_tag* and *max_tag*, sequentially to check whether or not the preceding RLE-character satisfies the query. As a result, the search I/O cost of the one-level SBC-tree is higher than that of the two-level SBC-tree.

6. EXPERIMENTAL RESULTS

In this section, we study experimentally the performance of the SBC-tree variants against the String B-tree that indexes uncompressed sequences.

Datasets: We conducted the experiments using three real datasets: SwissProt protein secondary structure database, Wal-Mart sales profile, and temperature readings from a sensor field. The SwissProt protein secondary structure database is available at *http://www.pir.uniprot.org/index.shtml* and consists of three values, i.e., $\Sigma = \{H = helix, S = strand, C = coil\}$. The Wal-Mart dataset contains sanitized data of timed sales transactions for several Wal-Mart stores. The dataset sequences consist of the hourly sales profiles discretized into five levels, i.e., $\Sigma = \{A = verylow, B = low, C = medium, D = high, E = veryhigh\}$. The temperature dataset consists of readings from a grid sensor field and is available at *http://dss.ucar.edu/*. The alphabet for the temperature dataset consists of 52 distinct integer values.

Query types: We measured the performance of the SBC-tree under four types of queries: *substring, prefix, range,* and *regular expression* queries. In the *regular expression* queries, the query pattern may contain frequency ranges, e.g., $X[i...j]$, which means that X appears from i to j times, or wild cards, e.g., $X*$, which means that X appears one or more times.

Query scenarios: Substring searching is a typical operation over biological databases. For example, given a protein segment s of unknown function, we want to search the database for all protein sequences that contain s. The results from this query can help biologists to infer the function

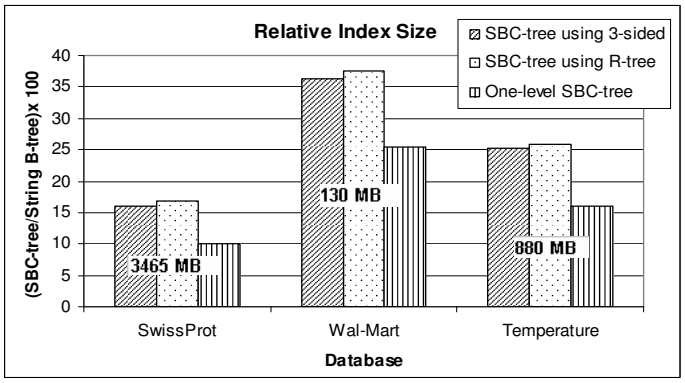


Figure 7: The index size

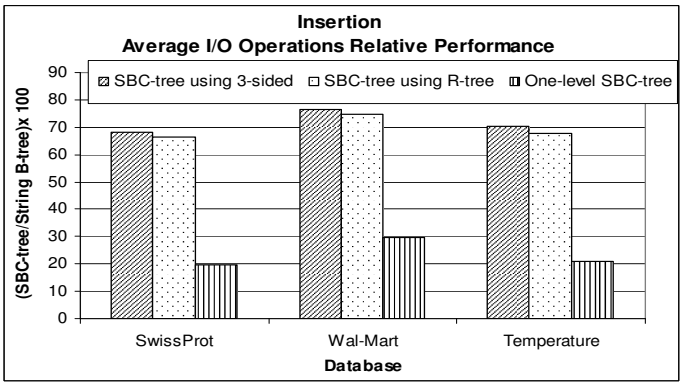


Figure 8: The *insert* operation

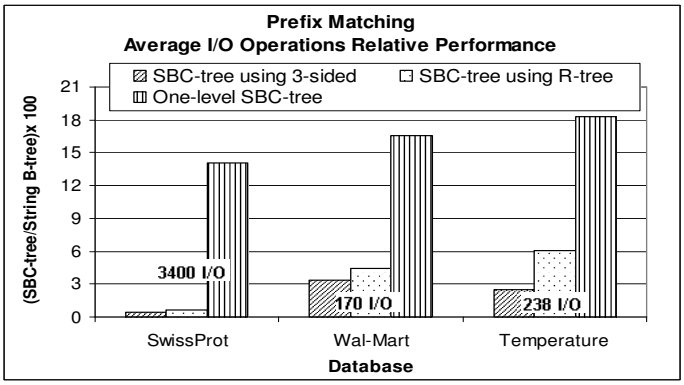


Figure 9: The *prefix matching* queries

and protein family of s. An example of a substring query over the protein database is: *SELECT * FROM PROTEINS WHERE SEQ* $\wedge\wedge$ *'H3S7C4'*; where *SEQ* is the sequence column that is indexed using the SBC-tree, and $\wedge\wedge$ is the *substring matching* operator. Regular expression searching is also an important operation over biological data, especially that biological sequences may have a degree of uncertainty and redundancy. An example of a regular expression query is: *SELECT * FROM PROTEINS WHERE SEQ* $\approx$ *'H[3...9]S7C4'*; where $\approx$ is the *regular expression* operator.

In time series databases, e.g., Wal-Mart and temperature readings datasets, although substring searching is not usually a direct operation over the data, it is used as a building block in many mining techniques that are commonly applied on these data sets. For example, in incremental frequent pattern mining techniques, the data items arrive to the database incrementally. A newly arrived item I may extend an already existing frequent pattern P to form another pattern PI that is candidate to be frequent. The data mining technique needs to search the database for PI to find out how many times PI appears in the database. A query example that retrieves the occurrences of the sales profile pattern $E5C1B2$ from Wal-Mart database is: *SELECT * FROM WAL-MART WHERE TIME-SERIES* $\wedge\wedge$ *'E5C1B2'*.

Query load: For each of the four query types, we generated several query patterns that range in length from 3 to 25 (uncompressed length). The size of the queries' answer set is inversely proportional to the length of the query patterns. The size of the answer set ranges from very few hits (less that 10) to many hits (thousands). The performance presented in the figures is the average of the queries' performances.

Performance results: In Figure 7, we present the SBC-tree index size relative to the String B-tree index size. The absolute String B-tree index size for each dataset is also presented in the figure. The figure illustrates that the one-level SBC-tree achieves up to an order of magnitude reduction in storage, and the SBC-tree using the 3-sided structure or the R-tree achieves up to 80% reduction in storage. The one-level SBC-tree involves the least storage overhead because it does not maintain a second-level index structure.

In Figure 8, we present the relative performance of the SBC-tree to insert all RLE-suffixes of a given sequence. The figure illustrates that the one-level SBC-tree achieves around 80% reduction in the number of I/Os, whereas, the SBC-tree using the 3-sided structure or the R-tree achieves around 30% saving in I/Os. This I/O saving is because all the SBC-tree variants index a small subset of the suffixes, i.e., the RLE-suffixes. The one-level SBC-tree achieves higher savings than the other SBC-tree variants because it does not require insertion in a second level structure.

In Figure 9, we present the SBC-tree I/O performance under *prefix matching* queries relative to the performance of the String B-tree. The absolute average number of I/O operations performed by the String B-tree is also presented in the figure. The SBC-tree using the 3-sided structure or the R-tree achieves around two orders of magnitude reduction in I/Os. The R-tree is a little worse than the 3-sided structure because the R-tree may involve traversing multiple paths in the tree. The one-level SBC-tree achieves less I/O saving than the two-level SBC-trees because the one-level SBC-tree scans the entire range specified by *min_tag* and *max_tag*, whereas the two-level SBC-trees applies a range query to retrieve the answer set.

Notice that, in the previous experiment, we treat suffixes that are prefixes to their sequences like all other suffixes. In order to achieve optimal I/O performance for answering *prefix matching* queries by both the String B-tree and the SBC-tree, we prefix each sequence in the database by a special character Ψ. In this case, all suffixes that are prefixes

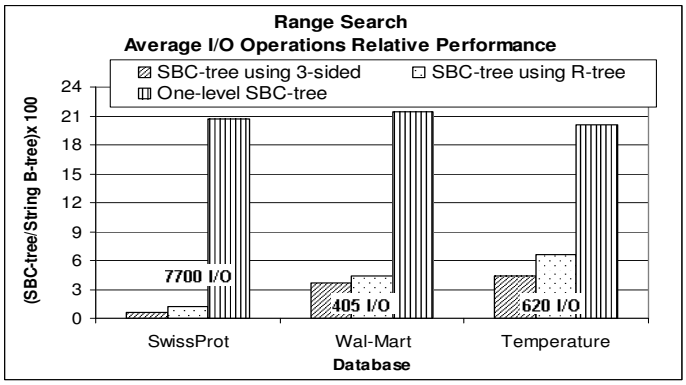


Figure 10: The *range search* queries

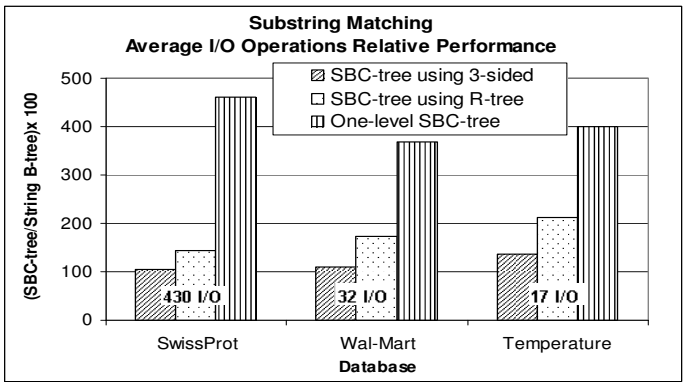


Figure 11: The *substring matching* queries

to their sequences are contiguous in the index tree. By prefixing the query pattern by Ψ, we guarantee that all leaf entries scanned by both the String B-tree and the SBC-tree belong to the query answer set. Therefore, the String B-tree and the SBC-tree can achieve the same optimal I/O performance.

The I/O performance of the SBC-tree under *range search* queries is presented in Figure 10. The absolute average number of I/O operations performed by the String B-tree is also presented in the figure. The figure illustrates that SBC-tree variants exhibit behavior similar to that of the *prefix matching* queries. The optimal I/O performance for answering *range search* queries can be reached by both the String B-tree and the SBC-tree in a manner similar to that in the case of the *prefix matching* queries.

The SBC-tree relative performance under *substring matching* queries is presented in Figure 11. The absolute average number of I/O operations performed by the String B-tree is also presented in the figure. The figure illustrates that the SBC-trees do not achieve I/O savings over the String B-tree of the uncompressed sequences. The reason is that the number of I/Os performed by the String B-tree is optimal, i.e., all leaf entries that are scanned by the String B-tree belong to the query answer set. The SBC-tree may require fewer number of I/Os to retrieve the same answer set since the sequences are compressed. However, accessing the two levels of the SBC-tree adds an extra I/O overhead.

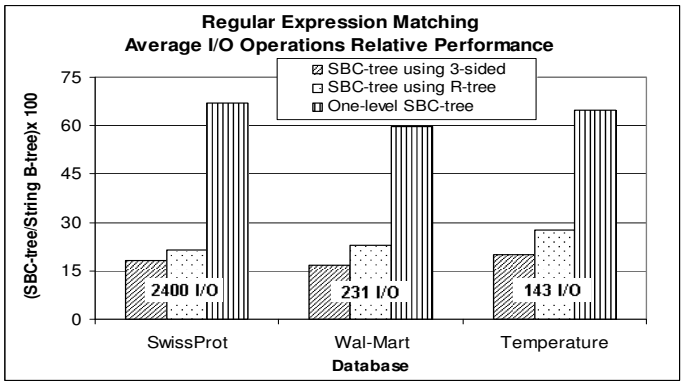


Figure 12: The *regular expression* queries

The SBC-tree using the 3-sided structure is the best among the SBC-tree variants. The R-tree involves higher I/O overhead than that of the 3-sided structure because the R-tree may traverse multiple paths in the tree. The one-level SBC-tree is the worst because it scans the range specified by the *min_tag* and *max_tag* sequentially to retrieve the answer set.

In Figure 12, we present the SBC-tree's relative performance under the *regular expression* queries. The figure illustrates that the SBC-tree achieves around 80% I/O saving over the String B-tree. The reason is that the String B-tree has to unfold the regular expression into multiple query patterns, e.g., *H[2...4]S5* will generate *H2S5, H3S5, and H4S5*, and then union the queries' answers, whereas the SBC-tree answers the query with no extra cost if the regular expression is at the beginning or the end of the query pattern. If the regular expression is at the middle of a query pattern, e.g., *H[2...4]S5E[1...10]S2*, then the SBC-tree will divide it into subpatterns that do not contain regular expression in the middle, e.g., P_1 =*H[2...4]S5* and P_2 = *E[1...10]S2*, and then union their answers, whereas the String B-tree will generate 13 query patterns and then union their answers.

In summary, the performance results illustrate that the SBC-tree achieves an optimal search performance over compressed sequences similar to that of the String B-tree over uncompressed sequences, with around 85% reduction in storage and 30% reduction in insertion I/Os.

7. CONCLUSION

We presented the SBC-tree index structure for indexing and searching RLE-compressed sequences of arbitrary length. The SBC-tree supports pattern matching queries such as *substring matching*, *prefix matching*, and *range search* operations over the compressed sequences. The SBC-tree has provable worst-case optimal theoretical bounds for the external-memory space requirements and search operations that are relative to the length of the compressed sequences. The structure is also dynamic and supports efficiently the insertion and deletion operations with provable amortized and worst-case theoretical bounds, respectively. We presented also a variant of the SBC-tree: the SBC-tree using the R-tree, that does not have provable worst-case theoretical bounds for search operations, but easier to realize inside current DBMSs and performs well in practice. Our perfor-

mance results illustrate that the SBC-tree achieves up to 85% reduction in storage, while retains the optimal search performance achieved by the String B-tree over the uncompressed sequences.

8. REFERENCES

[1] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: pattern matching in z-compressed files. In *SODA*, pages 705–714, 1994.

[2] A. Amir, G. Benson, and M. Farach. Optimal two-dimensional compressed matching. In *ICALP*, pages 215–226, 1994.

[3] A. Amir, G. M. Landau, and D. Sokol. Inplace run-length 2d compressed search. In *SODA*, pages 817–818, 2000.

[4] A. Amir, G. M. Landau, and U. Vishkin. Efficient pattern matching with scaling. *Journal of Algorithms*, 13(1):2–32, 1992.

[5] A. Apostolico, G. M. Landau, and S. Skiena. Matching for run-length encoded strings. *Journal of Complexity*, 15(1):4–16, 1999.

[6] O. Arbell, G. M. Landau, and J. S. Mitchell. Edit distance of run-length encoded strings. *Information Processing Letters*, 83(6):307–314, 2002.

[7] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *PODS*, pages 346–357, 1999.

[8] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[9] R. Bayer and K. Unterauer. Prefix b-trees. *TODS*, 2(1):11–26, 1977.

[10] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *VLDB Journal*, 5(4):264–275, 1996.

[11] S. J. Bedathur and J. R. Haritsa. Engineering a fast online persistent suffix tree construction. In *ICDE*, pages 720–731, 2004.

[12] T. Bell, M. Powell, A. Mukherjee, and D. Adjeroh. Searching bwt compressed text with the boyer-moore algorithm and binary search. In *DCC*, pages 112–121, 2002.

[13] H. Bunke and J. Csirik. Edit distance of run-length coded strings. In *Symposium on Applied computing*, pages 137–143, 1992.

[14] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.

[15] Y.-F. Chien, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed text indexing and range searching. Technical Report Purdue University tech. report, CSD TR06-021, DEC 2006.

[16] D. Comer. Ubiquitous b-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[17] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *STOC*, pages 365–372, 1987.

[18] P. Ferragina and R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *Journal of ACM*, 46(2):236–280, 1999.

[19] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398, 2000.

[20] W. B. Frakes and R. B. Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.

[21] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[22] V. Freschi and A. Bogliolo. Longest common subsequence between run-length-encoded strings: a new algorithm with improved parallelism. *Information Processing Letters*, 90(4):167–173, 2004.

[23] S. W. Golomb. Run-length encodings. *Trans. on Information Theory*, 12:399–401, 1966.

[24] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850, 2003.

[25] R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: experiments with compressing suffix arrays and applications. In *SODA*, pages 636–645, 2004.

[26] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA, 1997.

[27] E. Hunt, M. P. Atkinson, and R. W. Irving. A database index to large biological sequences. In *VLDB*, pages 139–148, 2001.

[28] R. W. Irving and L. Love. The suffix binary search tree and suffix avl tree. *JDA*, 1(5-6):387–408, 2003.

[29] V. Makinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. In *CMP*, pages 306–317, 2006.

[30] V. Makinen, G. Navarro, and E. Ukkonen. Approximate matching of run-length compressed strings. In *CPM*, pages 31–49, 2001.

[31] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal*, 22(5):935–948, 1993.

[32] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of ACM*, 23(2):262–272, 1976.

[33] E. M. McCreight. Priority search trees. *SIAM Journal*, 14(2):257–276, 1985.

[34] A. Moffat. Implementing the ppm data compression scheme. *Trans. on Communications*, 38(11):1917–1921, 1990.

[35] D. R. Morrison. Patricia: Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[36] G. Navarro. Regular expression searching on compressed text. *JDA*, 1(5-6):423–443, 2003.

[37] M. Patrascu and E. D. Demaine. Tight bounds for the partial-sums problem. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 20–29, 2004.

[38] N. S. Prywes and H. J. Gray. The organization of a multilist-type associative memory. In *Transactions on Communication and Electronics*, pages 488–492, 1963.

[39] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A model for sequence databases. In *ICDE*, pages

232–239, 1995.

[40] Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Pattern matching in text compressed by using antidictionaries. In *CPM*, pages 37–49, 1999.

[41] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column oriented dbms. In *VLDB*, 2005.

[42] H. Tanaka and A. L. Garcia. Efficient run-length encodings. *Trans. on Information Theory*, 28(6):880–889, 1982.

[43] S. Tata, R. A. Hankins, and J. M. Patel. Practical suffix tree construction. In *VLDB*, pages 36–47, 2004.

[44] T. E. Tzoreff. Matching patterns in strings subject to multi-linear transformations. *TCS*, 60(3):231–254, 1988.

[45] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *TKDE*, 9(3):391–409, 1997.

[46] J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE DATA. *ACM Computing Surveys*, 33(2):209–271, 2001.

[47] P. Weiner. Linear pattern matching algorithms. In *Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[48] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *Trans. on Information Theory*, 23(3):337–343, 1977.

[49] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *Trans. on Information Theory*, 24(5):530–536, 1978.