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

ABSTRACT

Run-Length-Encoding (RLE) is a data compression technique that is used in various applications, e.g., biological sequence databases, multimedia, and facsimile transmission. One of the main challenges is how to operate, e.g., indexing, searching, and retrieval, on the compressed data without decompressing it. In this paper, we present the String B-tree for Compressed 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. The SBC-tree supports *substring* as well as *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. The insertion and deletion of all suffixes of a compressed sequence of length m takes $O(m \log_B(N+m))$ I/O operations. *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. We present also two variants of the SBC-tree: the SBC-tree that is based on an R-tree instead of the 3-sided structure, and the one-level SBC-tree that does not use a two-dimensional index. These variants do not have provable worst-case theoretical bounds for search operations, but perform well in practice. The SBC-tree index is realized inside PostgreSQL in the context of a biological protein database application. Performance 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.

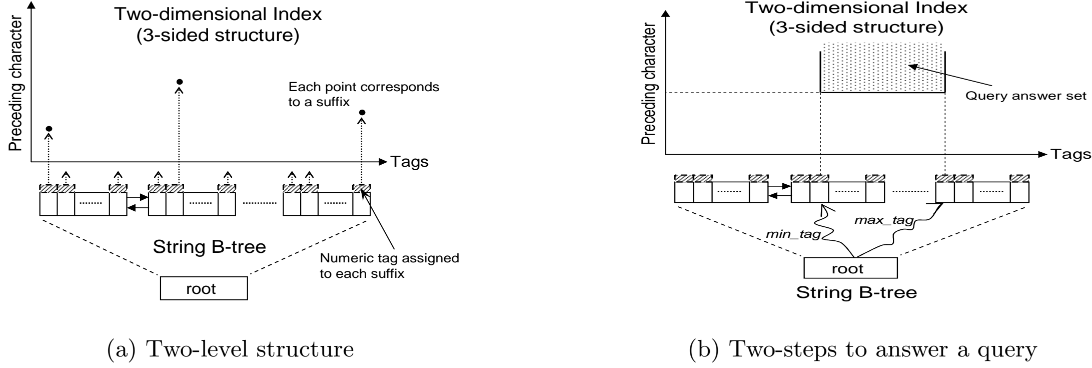
1. INTRODUCTION

Current databases store massive amounts of data, especially in text and sequence formats, e.g., biological sequences, text books, medical record, multimedia files, digital libraries, etc. With such massive amounts of data, data compression techniques, e.g., [15, 24, 36, 42, 48, 49], gain significant importance to achieve compact data representation. Run-Length-Encoding (RLE) [24] 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 in various applications, e.g., biological sequence databases, multimedia, and facsimile transmission.

One of the main challenges is how to operate, e.g., indexing, searching, and retrieval, on the compressed data without decompressing it. The goal is to achieve search performance over the compressed data that is better than or at least competitive with the search performance over the uncompressed data. Several in-memory algorithms have been proposed to search various formats of compressed data, e.g., [1, 2, 6, 13, 14, 20, 23, 26, 32, 41]. However, none of the proposed algorithms address the problem of indexing and searching compressed data using external memory techniques [46].

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(a). The first level is a modified version of the String B-tree proposed in [19], and the second level is the optimum 3-sided range query structure proposed in [8]. The 3-sided structure is built on top of the leaf entries of the modified String B-tree. For each suffix k in sequence S inserted into the modified String B-tree, a point containing k's preceding character in S and a tag indicating k's position in the String B-tree is inserted into the 3-sided structure. The link that relates each suffix in the modified String B-tree to a point in the 3-sided structure is the tag value.

The SBC-tree supports *substring* as well as *prefix matching* and *range search* queries over RLE-compressed sequences. A query over the SBC-tree is answered in two steps (See Figure 1(b)). In the first step, the SBC-tree first level, i.e., the String B-tree, determines a range, specified by two tag values *min_tag* and *max_tag*, that contains a superset of the query answer. This range is mapped in the second step into a two-dimensional range query over the SBC-tree second



(a) Two-level structure

(b) Two-steps to answer a query

Figure 1: The SBC-tree design

level to retrieve only the required answer set.

We formalize our problem as follows. Given a set of K RLE-compressed sequences $\Delta = \{S_1, S_2, ..., S_K\}$, where S_i is a sequence of length n in the form $S_i =$ '$x_1 f_1 \ x_2 f_2 \ ... \ x_n f_n$', x_j is a character in the alphabet Σ, and $f_j \geq 1$ is the frequency of x_j. We call $x_j f_j$ an *RLE-character*. Sequence S_i has n *RLE-suffixes*, i.e., RLE-Suffixes(S_i) $= \{x_j f_j \ x_{j+1} f_{j+1} \ ... \ x_n f_n \mid 1 \leq j \leq n\}$. The length of the decompressed sequence of S_i is the sum of the character frequencies forming S_i. That is, $|decompressed(S_i)| = \sum_{j=1}^{n} f_j$, which can be much larger than n. The decompressed sequence of S_i has $\sum_{j=1}^{n} f_j$ suffixes. The n RLE-suffixes of S_i 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. Using the proposed SBC-tree, we achieve 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, and (3) efficiently answer pattern 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 takes $O(m \log_B (N+m))$ 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.

We also present two variants of the SBC-tree: the SBC-tree that is based on an R-tree instead of the 3-sided structure, and the one-level SBC-tree that does not use a two-dimensional index. These variants do not have provable worst-case theoretical bounds for search operations, but perform well in practice.

The contributions of this paper are summarized as follows:

1. We present an index structure, termed the SBC-tree, for indexing and searching RLE-compressed sequences. The SBC-tree is realized inside PostgreSQL.
2. The SBC-tree has provable worst-case optimal theoretical bounds for the external-memory space requirements, and search operations.
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. To the best of our knowledge, this paper is the first to address indexing compressed data in external memory.

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, namely, the modified String B-tree and the 3-sided range query structure. We present the SBC-tree structure along with its update and search algorithms in Sections 4 and 5. The theoretical analysis and experimental results of the SBC-tree are presented in Sections 6, and 7, respectively. Section 8 contains concluding remarks.

2. RELATED WORK

The concept of searching compressed data was introduced in [5, 44]. Based on this concept, several in-memory algorithms have been proposed to search various formats of compressed data. Algorithms for searching RLE-compressed sequences include substring matching [3, 4, 44], approximate pattern matching [32], edit distance [7, 14], and longest common subsequence [6, 23]. Algorithms over other compression schemes include searching Lempel-Ziv compressed data [2, 38], searching antidictionaries compressed text [41], and searching Burrows-Wheeler transform (BWT) compressed data [13]. Several in-memory algorithms and data structures have been proposed to search entropy compressed text [20, 25, 26]. For applications such as entropy compressed text, the encoding scheme is complex, and hence the search mechanisms and compression formats have to be carefully engineered. For this purpose, several in-memory pattern matching data structures are proposed that are based on Burrows Wheeler Transform(BWT) [20] and Compressed Suffix Arrays(CSA) [25, 26]. However, indexing and searching compressed data in external memory is more challenging, and no external memory structures analogous to the structures above exist.

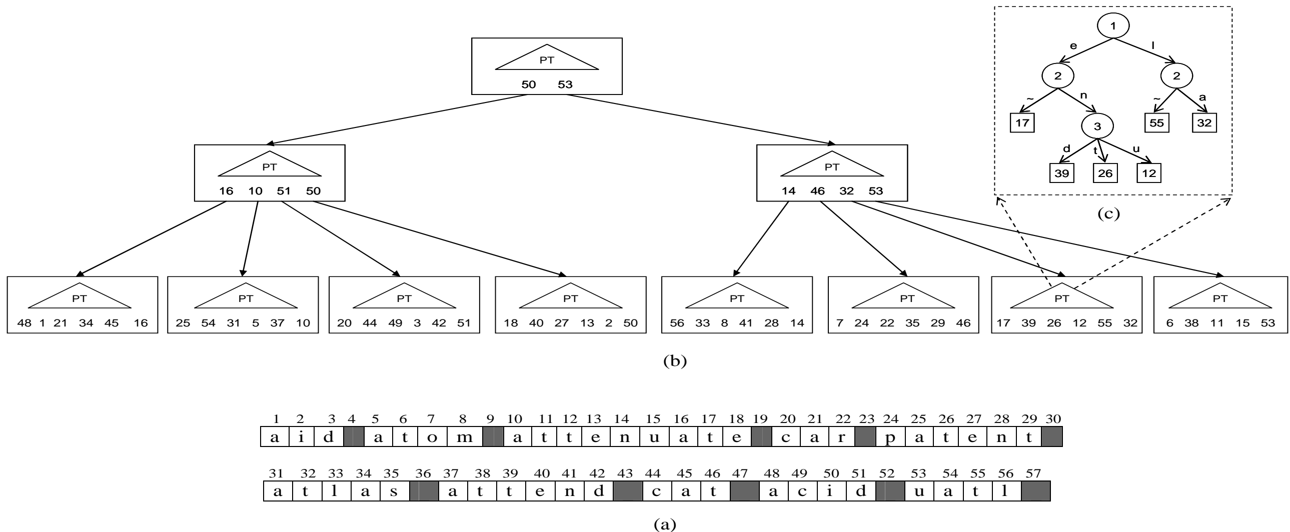


Figure 2: (a) Strings on disk, (b) The String B-tree for all suffixes, (c) The Patricia trie inside one node.

Regardless of the compression technique used, compressed sequences are usually treated as string sequences. Therefore, indexing compressed sequences is closely tied to text and sequence indexing. A model for sequence databases, called *SEQ*, is proposed in [40]. SEQ models different types of sequence data and defines a set of operators to query the sequences. A data structure for indexing numeric sequences is proposed in [18], where sequences are mapped into rectangles and indexed using multidimensional access methods. Several well-known index structures for text indexing have been proposed. These structures include suffix trees [27, 34, 47], suffix binary search trees [31], suffix arrays [21, 27, 33], inverted files [39], tries [22, 37], B-trees [9, 16], and the prefix B-tree [10]. Several variants of these structures have been proposed to index efficiently strings of unbounded length. The persistent suffix trees have been proposed in [12, 30]. A buffer management strategy for a practical construction of suffix trees has been proposed in [43]. An external memory structure for suffix arrays in a form of B-tree is the String B-tree [19]. The String B-tree is a combination of the B-tree and the Patricia trie that is used for I/O-efficient searching on strings of arbitrary length.

Using existing text indexing data structures to index RLE-compressed sequences is not straightforward. The reason is that the structure and search mechanisms of the existing indexes are based on storing all suffixes of the underlying sequences, which is not the case in indexing the RLE-compressed sequences, where the indexed RLE-suffixes are only a small subset of the sequences' total suffixes. 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.

One of the SBC-tree applications is indexing protein secondary structure sequences. Protein secondary structure sequences can be highly compressed using RLE compression technique. Indexing and searching protein secondary structure sequences is addressed in [29], where a new query language and a set of algorithms are proposed to search protein sequences. The algorithms proposed in [29] are for indexing and searching the uncompressed sequences format.

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 [19] is a data structure for indexing strings of arbitrary length, where the 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 the suffixes' 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 [37], where the entries inside each B-tree node are organized in a Patricia trie structure instead of a sequential array (See Figure 2(c)). The goal of the Patricia trie is to avoid the logarithmic search inside each B-tree node. The reason is that each comparison against an index key requires performing one I/O to retrieve the key data from the disk. Using the Patricia trie, we can avoid the logarithmic search and perform only one I/O per B-tree node instead of $\log_2 n$ I/Os, where n is the number of entries in the B-tree node. For example, consider searching for pattern "tlas" in the node highlighted in Figure 2(c). If the keys inside the node are stored sequentially, then the binary search involves three comparisons with logical keys 26, 55, and 32, which requires three I/Os to get the keys' data. In contrast, searching the Patricia trie (Figure 2(c)) is performed by following the branching character l followed by a to reach logical key 32. Then we perform one I/O to get the data corresponding to this logical key. The exact location of the searched pattern inside the node can then be specified based on the comparison between the pattern and the retrieved key data without further I/Os.

In Figure 2, we illustrate 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 (assuming the start position is 0), and the branching characters are e and l.

To support efficiently the insertion of the strings' suffixes, the String B-tree maintains two types of auxiliary pointers. The standard *parent pointer* defined for each node, and the *succ pointer* defined for each key in the index. The succ pointer of the key corresponding to suffix k in string S points to the index node containing the key corresponding to suffix $k+1$ in S. The succ pointer of the last suffix in a string points to itself. Using such auxiliary pointers, we perform a root-to-leaf path traversal in the index tree only for inserting the first suffix in each string. Then all the succeeding suffixes are inserted by following the auxiliary pointers.

Searching the String B-tree is done by performing 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. For example, in Figure 2, *substring searching* for pattern $p =$ "en" proceeds by performing a root-to-leaf path to locate the lexicographically first suffix containing p, which is the suffix starting at position 40 on the disk. Another root-to-leaf path is performed to locate the lexicographically last suffix containing p, which is the suffix starting at position 13 on the disk. All suffixes inbetween these two keys satisfy the query, i.e., suffixes starting at positions $40, 27$, and 13.

The String B-tree has good performance in answering pattern matching queries and has worst-case theoretical bounds like the ones of the regular B-tree. The following lemma states the theoretical bounds of the String B-tree [19].

Lemma 1. :

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).

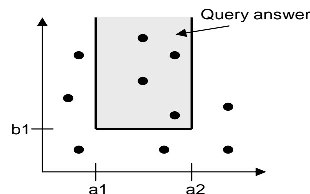


Figure 3: 3-sided query ($a1$, $a2$, $b1$).

The 3-sided range query structure [8] is an external memory structure that is based on the external memory priority search tree [35] and the persistent B-tree [11, 45]. 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 [8].

Lemma 2. :

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.

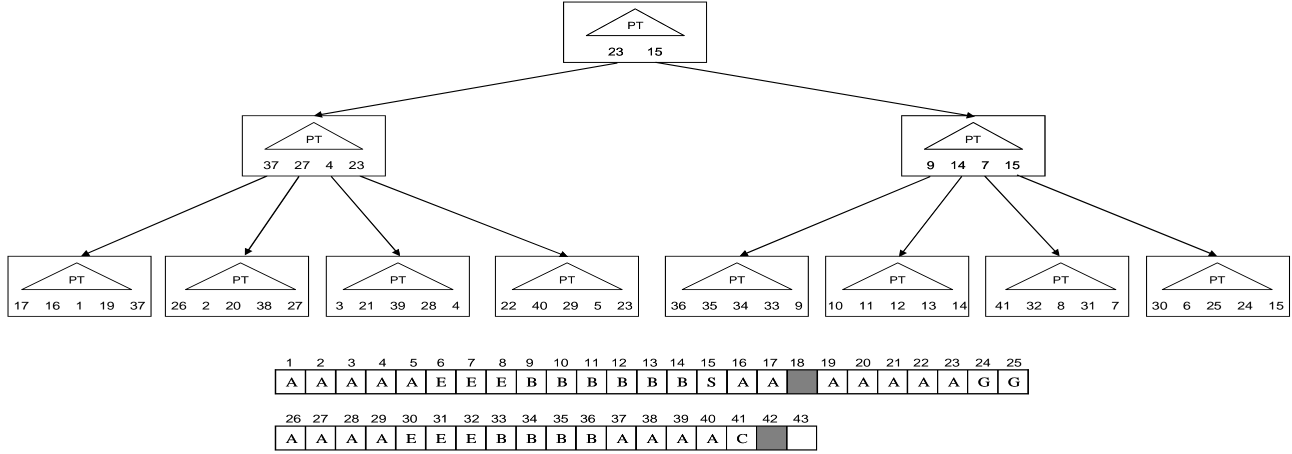
c) The 3-sided range query executes in $O(\log_B N + \frac{T}{B})$ worst-case I/O operations, where T is the output size.

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. 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 $p = A2E3B4$ over both indexes. The search over the uncompressed suffixes (Figure 4(a)) 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)) 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.

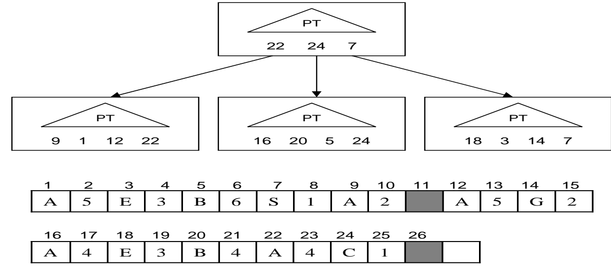
The trick to answer the *substring matching* query correctly over the RLE-suffixes is to map the query pattern



(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

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

$p = A2E3B4$ 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 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.

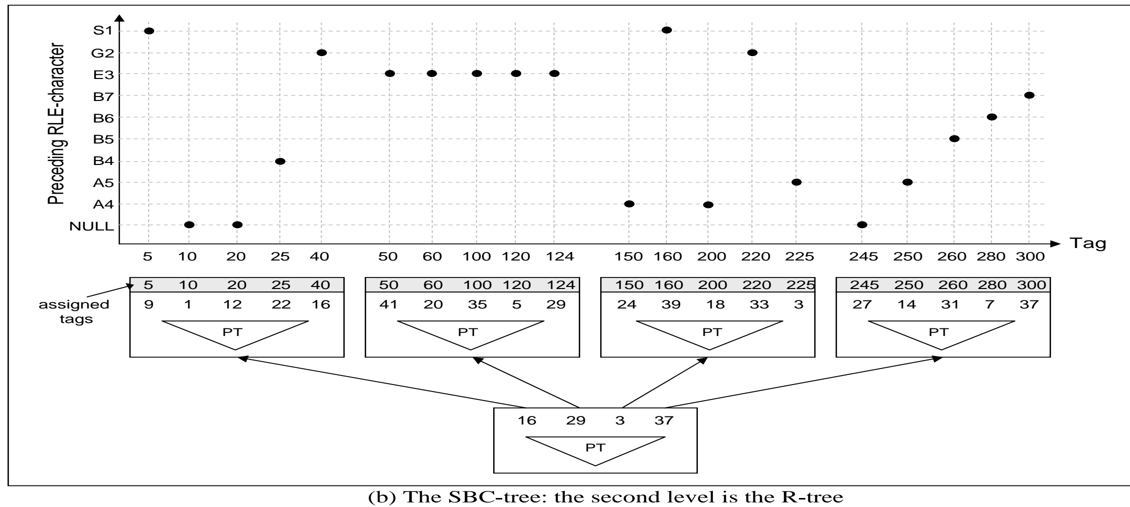
Rule 1. A *substring matching* query pattern $p = x_1f_1\ \ x_2f_2\ \ ...\ \ x_nf_n$ over RLE-suffixes is mapped into pattern $p' = x_1f_1^{+}\ \ x_2f_2\ \ ...\ \ x_nf_n$, where $x_1f_1^{+}$ means repeats of character x_1 of length larger than or equal to f_1.

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 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. For example, the two RLE-suffixes $A5E3B6S1A2$ and $A4E3B4A4C1$ starting at positions 1 and 16, respectively, on the disk (See Figure 4(c)) satisfy the query pattern $p' = A2^{+}E3B4$. 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 a two-dimensional index structure. The first level of the SBC-tree stores the RLE-suffixes of the inserted RLE-compressed sequences. The second level of the SBC-tree stores for each inserted RLE-suffix a reference to that suffix, i.e., a tag, and the suffix's preceding RLE-character in the suffix's sequence (See Figure 5).

The modification we introduce to the String B-tree is a numeric tag assigned to each leaf entry that reflects the entry's position in the index. 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 the order-maintenance technique [17]. We discuss the assignment of the tags in detail in Section 5.2. The second level of the SBC-tree can be any two-dimensional index structure. We



(b) The SBC-tree: the second level is the R-tree

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A	5	E	3	B	6	S	1	A	2		A	5	G	2	A	4	E	3	B	4

22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
A	4	C	1		E	3	B	5	G	2	E	3	B	7	S	1	E	3	B	4

(a) The RLE sequences on the disk

Figure 5: The SBC-tree for sequences $S_1 = A5E3B6S1A2$, $S_2 = A5G2A4E3B4A4C1$, and $S_3 = E3B5G2E3B7S1E3B4$.

consider, in this paper, the use of the R-tree [28] and the 3-sided range query structure [8]. The R-tree has a good performance in practice and is available in several DBMSs, however, no theoretical bounds are guaranteed. The 3-sided range query structure, on the other hand, has an optimal theoretical bound for the two-dimensional range searching. In this section, we use the R-tree as the second level of the SBC-tree. In Section 5.1, we discuss the use of the 3-sided range query structure instead of the R-tree.

The SBC-tree is maintained during the insertion as follows (an example is illustrated in Figure 5). The insertion of an RLE-compressed sequence $S = x_1 f_1 \; x_2 f_2 \; ... \; x_n f_n$ is performed as follows.

1. Insert the RLE-suffixes of S into the String B-tree.
2. For each inserted RLE-suffix, e.g., $x_i f_i \; x_{i+1} f_{i+1} \; ... \; x_n f_n$, where $1 \leq i \leq n$, maintain two attributes:
   (a) The numeric tag assigned to the suffix.
   (b) The suffix's preceding RLE-character, i.e., $x_{i-1} f_{i-1}$. If the inserted suffix is the first suffix of S, i.e., $i = 1$, then the preceding RLE-character is NULL.
3. Insert the two attributes, i.e., the tag and the preceding RLE-character, as a point into the second level of the SBC-tree. The RLE-suffix's position on the disk is attached to the point.

In Figure 5, we illustrate the structure of the SBC-tree for sequences $S_1 = A5E3B6S1A2$, $S_2 = A5G2A4E3B4A4C1$, and $S_3 = E3B5G2E3B7S1E3B4$. The *preceding character* dimension of the R-tree is ordered alphabetically, where the *NULL* character is considered as the first character in the alphabet. Each RLE-suffix has a corresponding point in the R-tree. For example, the first RLE-suffix in the String B-tree ($A2$ in sequence S_1 with tag 5) is preceded by the RLE-character $S1$ on the disk. Therefore, the corresponding point to this suffix is (5, $S1$) in the R-tree.

In the following sections we discuss how the SBC-tree is used to answer the *substring matching*, *prefix matching*, and *range search* queries.

4.2 Answering Substring Matching Queries

Query Definition: Given a query pattern p, where $p = x_1 f_1 \; x_2 f_2 \; ... \; x_n f_n$, find all substrings in the database whose prefix matches p.

A *substring matching* query is answered as follows.

1. If the length of p is 1, i.e., $p = x_1 f_1$, then execute Step 2, else execute Steps 3 to 5.
2. Search the SBC-tree first level, i.e., the String B-tree, for p. The answer from the String B-tree is a 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 prefix matches p, respectively. All the RLE-suffixes between *min_tag* and *max_tag* are the answer to the *substring matching* query.
3. Map the query pattern p, according to Rule 1, into $p' = x_1 f_1^+ \; x_2 f_2 \; ... \; x_n f_n$.
4. Search the SBC-tree first level, i.e., the String B-tree, for pattern $p'' = x_2 f_2 \; ... \; x_n f_n$, ignoring the first RLE-character $(x_1 f_1^+)$ in p'. The answer from the String B-tree is a 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 prefix matches p'', respectively.

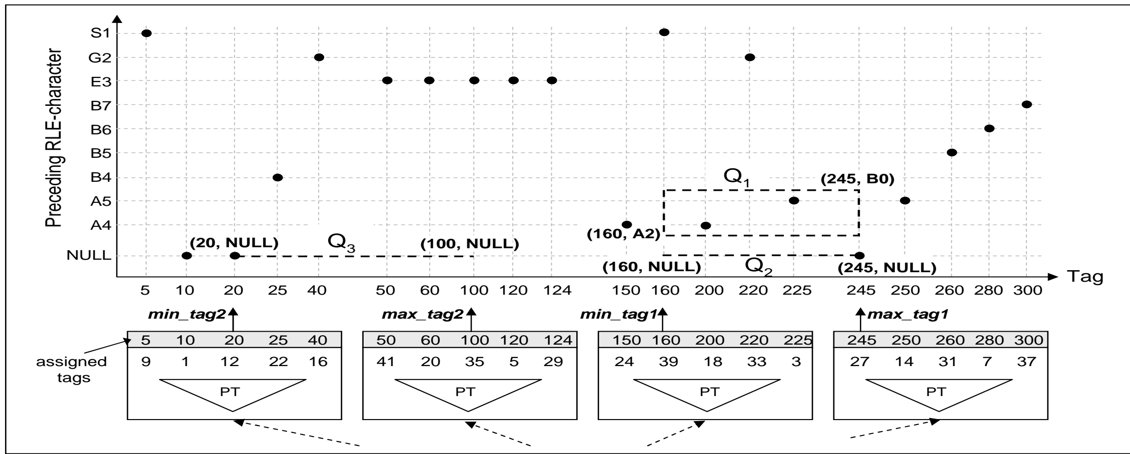


Figure 6: Searching the SBC-tree. Substring matching Q_1, Prefix matching Q_2, Range query Q_3.

5. Apply a two-dimensional range query over the SBC-tree second level, i.e., the R-tree, where the *tag* dimension ranges from *min_tag* to *max_tag*, and the *preceding character* dimension ranges from $x_1 f_1$ to $\varphi 0$, where φ is x_1's succeeding character in the alphabet Σ. The answer to the range query is the answer to the *substring matching* query.

In Step 1, if the length of p is 1, then the query answer is contiguous inside the String B-tree. Therefore, we retrieve in Step 2 all the RLE-suffixes between the specified *min_tag* and *max_tag* values as the answer to the query. If the length of p is larger than 1, then we execute Steps 3 to 5. In Step 3, we map p to p' in order to retrieve the RLE-suffixes whose prefix matches p and implicit-suffixes whose prefix matches p. In Step 4, we search for pattern p'' instead of p' because the RLE-suffixes whose prefix matches p' are not guaranteed to be contiguous inside the String B-tree index. However, the RLE-suffixes whose prefix matches p'' are a superset of the required answer and they are contiguous inside the String B-tree index. Therefore, we can locate the first and last RLE-suffixes satisfying p'' without enumerating the suffixes in-between. In Step 5, we filter the answer superset by retrieving only the RLE-suffixes whose preceding RLE-character satisfies $x_1 f_1^+$, i.e., the RLE-suffixes that satisfy p'. The exact start position of the suffixes satisfying p' can be easily computed from the answer to the two-dimensional range query.

In Figure 6, we give an example of *substring match* searching for pattern $p = A2E3B4$. The corresponding p' and p'' will be $A2^+E3B4$ and $E3B4$, respectively. The search for p'' over the String B-tree returns the two tags *min_tag1*= 160 and *max_tag1*= 245. The range query over the R-tree, denoted by Q_1, has bottom-left and top-right coordinates of $(160, A2)$ and $(245, B0)$, respectively. The answer to the range query is the two RLE-suffixes starting at positions 18 and 3 on the disk (Figure 5(a)). By subtracting the length of the RLE-characters preceding those suffixes, e.g., $A4$ and $A5$ have length of 2, we get the exact start position of the RLE-suffixes satisfying p, i.e., the suffixes starting at positions 16 and 1 on the disk. Notice that the suffixes at positions 16 and 1 are not contiguous in the String B-tree.

4.3 Answering Prefix Matching Queries

Query Definition: Given a query pattern p, where $p = x_1 f_1 \; x_2 f_2 \; ... \; x_n f_n$, 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, no implicit-suffixes can be an answer to the query because implicit-suffixes are not prefixes to their sequences. Therefore, we do not need to apply the mapping rule (Rule 1) to pattern p.

A *prefix matching* query is answered as follows.

1. Search the SBC-tree first level, i.e., the String B-tree, for pattern $p = x_1 f_1 \; x_2 f_2 \; ... \; x_n f_n$. The answer from the String B-tree is a 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 prefix matches p, respectively.

2. Apply a two-dimensional range query over the SBC-tree second level, i.e., the R-tree, where the *tag* dimension ranges from *min_tag* to *max_tag*, and the *preceding character* dimension equals NULL. The answer to the range query is the answer to the *prefix matching* query.

In Figure 6, we give an example of *prefix match* searching for pattern $p = E3B4$. The search for p over the String B-tree returns the two tags *min_tag1*= 160 and *max_tag1*= 245. The two-dimensional range query over the R-tree, denoted by Q_2, has bottom-left and top-right coordinates of $(160, NULL)$ and $(245, NULL)$, respectively. The answer to the range query is one RLE-suffix that starts at position 27 on the disk (Figure 5(a)). All the other suffixes in the range are not prefixes to their sequences.

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.

In *range search* queries, suffixes that satisfy the query have to be prefixes to their sequences, i.e., the suffix is the entire

sequence. Therefore, we do not need to apply the mapping rule (Rule 1) to patterns p_1 and p_2.

A *range search* query is answered as follows.

1. Search the SBC-tree first level, i.e., the String B-tree, to locate the first key larger than or equal to p_1. Similarly, search the String B-tree to locate the last key smaller than or equal to p_2. The answer from the String B-tree is a range specified by two tags, *min_tag* and *max_tag*.

2. Apply a two-dimensional range query over the SBC-tree second level, i.e., the R-tree, where the *tag* dimension ranges from *min_tag* to *max_tag*, and the *preceding character* dimension equals NULL. The answer to the range query is the answer to the *range search* query.

The range specified by *min_tag* and *max_tag* in Step 1 includes the RLE-suffixes whose lexicographic order is between p_1 and p_2. In Step 2, we filter this range by retrieving only the RLE-suffixes that are prefixes to their database sequences, i.e., the preceding RLE-character is NULL.

In Figure 6, we give an example of a *range search* query, where $p_1 = A5G1$ and $p_2 = B7S2$. The first RLE-suffix larger than or equal to p_1 is the suffix starting at position 12 on the disk, i.e., *min_tag* = 20. The last RLE-suffix smaller than or equal to p_2 is the suffix starting at position 35 on the disk, i.e., *max_tag* = 100. The two-dimensional range query over the R-tree, denoted by Q_3, has bottom-left and top-right coordinates of $(20, NULL)$ and $(100, NULL)$, respectively. The answer to the range query is one RLE-suffix that starts at position 12 on the disk (Figure 5(a)). All the other suffixes in the range are not prefixes to their sequences. Notice that the suffix starting at position 1 on the disk, i.e., $A5E3B6S1A2$, includes implicit-suffixes that are between p_1 and p_2 in lexicographic order, e.g., $A4E3B6S1A2$ and $A3E3B6S1A2$. However, these suffixes are not prefixes to their sequences.

5. THE SBC-TREE DESIGN ISSUES

5.1 The Use of the 3-sided Structure

Although the R-tree has a good performance in practice, the worst-case theoretical bounds for the update and search operations are not guaranteed. In this section, we discuss using the 3-sided range query structure proposed in [8] as the SBC-tree second level. The 3-sided range query structure has an optimal worst-case theoretical bound for the 3-sided two-dimensional range queries as discussed in Section 3.2. By using the 3-sided range query structure, we can achieve the SBC-tree claimed theoretical bounds.

The key point is that instead of maintaining one R-tree structure for all characters in the alphabet Σ, we maintain a separate 3-sided range query structure for each character in Σ. We then insert each point in the space into the appropriate 3-sided structure based on the point's *preceding character* dimension. In Figure 7, we illustrate the SBC-tree using the 3-sided structure for sequences illustrated in Figure 5(a). We maintain separate 3-sided structures for the characters $NULL, A, B, E, G$, and S. The $NULL$ character is a special character in that its structure is a one-dimensional structure, i.e., the *preceding character* dimension contains only one value, the $NULL$ value. Therefore, the B-tree can be used to index the points belonging to this structure.

To answer a *substring matching* query for pattern $p' = x_1f_1^+ \ x_2f_2 \ ... \ x_nf_n$, we map the range obtained from the String B-tree and specified by the *min_tag* and *max_tag* values (See Section 4.2) into a 3-sided query over the structure corresponding to character x_1. The *tag* dimension ranges from *min_tag* to *max_tag*, and the *preceding character* dimension is larger than or equal to x_1f_1. In Figure 7, we give an example of *substring match* searching for pattern $p' = A2^+E3B4$. The *min_tag* and *max_tag* specified by the String B-tree are 160 and 245, respectively. The 3-sided query, denoted by Q_1, has the *tag* dimension ranges from 160 to 245, and the *preceding character* dimension is larger than or equal to $A2$. The answer to the query is the two suffixes starting at positions 18 and 3 on the disk.

To answer *prefix matching* or *range search* queries, we map the range obtained from the String B-tree and specified by the *min_tag* and *max_tag* values (See Sections 4.3 and 4.4) into a one-dimensional query over the structure corresponding to the $NULL$ character. We illustrate in Figure 7, a *prefix match* searching for pattern $p = E3B4$. The *min_tag* and *max_tag* specified by the String B-tree are 160 and 245, respectively. The corresponding range query, denoted by Q_2, returns one suffix as the answer to the query, i.e., the suffix starting at position 27 on the disk.

5.2 The SBC-tree Tags Assignment

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 second level and re-inserted with the new tag values.

Dietz and Sleator [17] propose a scheme 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 points corresponding to these entries in the SBC-tree second level will be deleted and re-inserted. The scheme proposed in [17] does not require any data structure other than the tags stored inside the String B-tree, and has the property that the re-assigned tags are in a contiguous region. Thus, the point updates in the two-dimensional space are within a contiguous *tag-dimension* range. Therefore, the 3-sided structure deletes and re-inserts the $\log_2 N$ points in $O(\log_B N + (\log_2 N)/B) = O(\log_B N)$ I/O operations. The following lemma states the complexity of tag assignment.

Lemma 3. Assigning a tag to a newly inserted RLE-suffix and updating the 3-sided structure accordingly takes an amortized $O(\log_B N)$ I/O operations.

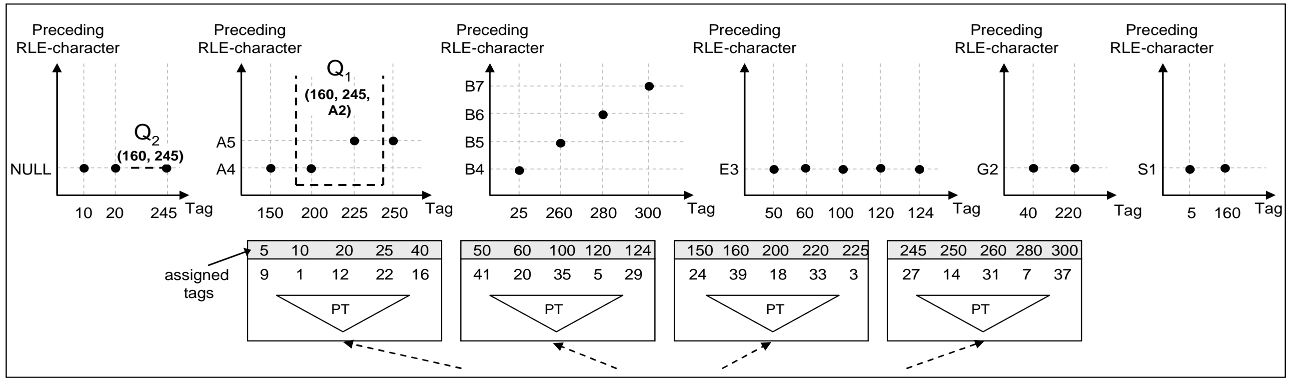


Figure 7: The SBC-tree: the second level is the 3-sided structure

Parameter	Definition
B	The disk page size
N	The total length of the RLE-compressed sequences. Also, the number of points in the 2D space
T	The query output size
$\|p\|$	The length of a RLE-compressed query pattern
m	The length of an RLE-compressed sequence to be inserted or deleted
$\|\Sigma\|$	The alphabet size

Table 1: The analysis parameters

5.3 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. In the experiments section (Section 7) we compare the two-level and one-level SBC-tree variants and illustrate their advantages and disadvantages.

6. THEORETICAL ANALYSIS

In this section, we present an analysis of the SBC-tree update and search operations. We consider the SBC-tree as described in Section 5.1, i.e., the SBC-tree using the 3-sided structure. We derive the SBC-tree theoretical bounds from the theoretical bounds of the String B-tree and the 3-sided range query structures. The parameters used in the analysis are summarized in Table 1.

6.1 Space Requirement

The SBC-tree structure consists of a single String B-tree, and $|\Sigma|$ 3-sided structures. The space complexity of the String B-tree is $O(N/B)$ pages (Lemma 1a), and the combined space complexity of the $|\Sigma|$ 3-sided structures is $O(N/B)$ pages (Lemma 2a). Based on these bounds, we derive the following lemma.

Lemma 4. The SBC-tree has an optimal external-memory space complexity of $O(N/B)$ pages.

6.2 Update

The insertion of an RLE-compressed sequence of length m requires (1) inserting m suffixes into the String B-tree that requires $O(m \log_B(N+m))$ I/O operations (Lemma 1b), (2) assigning m tags to the inserted RLE-suffixes, and possibly updating the 3-sided structure in the case of tag reassignment, that takes $O(m \log_B(N+m))$ amortized I/O operations (Lemma 3), and (3) inserting m points into the 3-sided structure that requires $O(m \log_B(N+m))$ I/O operations (Lemma 2b). The deletion of an RLE-compressed sequence of length m requires (1) deleting m suffixes from the String B-tree that executes in $O(m \log_B(N+m))$ I/O operations (Lemma 1b), and (2) deleting m points from the 3-sided structure that executes in $O(m \log_B(N+m))$ I/O operations (Lemma 2b). Based on these bounds, we derive the following lemma.

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.

6.3 Search

Substring matching, *prefix matching*, and *range search* queries over the SBC-tree are answered by performing (1) two root-to-leaf path traversals over the String B-tree that execute in $O(\log_B N + \frac{|p|}{B})$ I/O operations (Lemma 1c), and (2) a 3-sided range query over the 3-sided structure that executes in $O(\log_B N + \frac{T}{B})$ I/O operations (Lemma 2c). Based on these bounds, we derive the following lemma.

Lemma 6. Substring matching, *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 bound for the *prefix matching* and *range search* queries is optimal under the assumption that indexing all suffixes is required to answer the *substring matching*

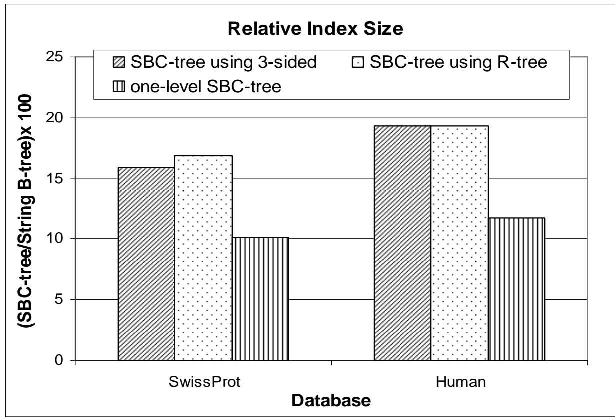


Figure 8: The index size

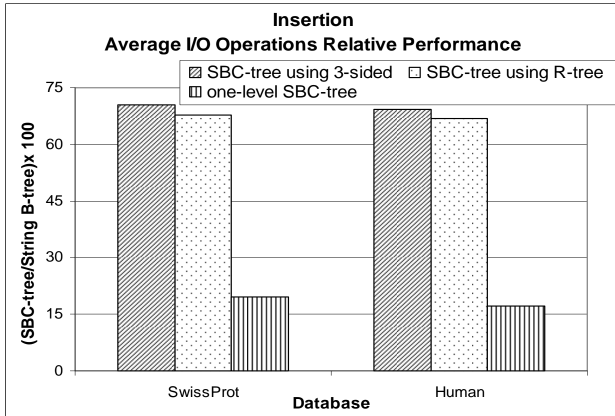


Figure 9: The performance under *insert* operations

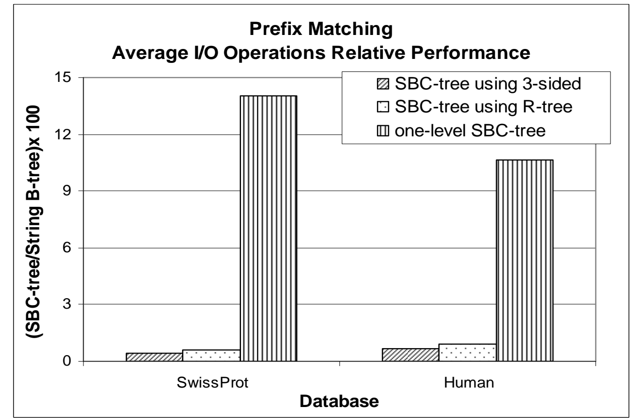


Figure 10: The performance under *prefix matching* queries

queries. If *substring matching* is not of interest, then 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 [19].

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 all 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.

7. EXPERIMENTAL RESULTS

In this section, we study experimentally the performance of the SBC-tree in the context of protein secondary structure databases. We use the Human and SwissProt protein databases available at *http://www.pir.uniprot.org/index.shtml.* These databases are among the largest protein databases available online. The String B-tree index size for the uncompressed sequences of the SwissProt and Human databases is 3.5 GB and 1.2 GB, respectively. The alphabet for the protein sequences consists of three letters, i.e., $\Sigma = \{E, H, L\}$, and the sequences consist of long repeats of these letters.

In our experiments, we study the performance of the SBC-tree for indexing RLE-compressed sequences against the String B-tree for indexing the uncompressed sequences. We consider three variants of the SBC-tree, the SBC-tree using the 3-sided structure, the SBC-tree using the R-tree, and the one-level SBC-tree.

In Figure 8, we present the SBC-tree index size relative to the String B-tree index size. The figure illustrates that the one-level SBC-tree achieves around an order of magnitude reduction in storage, and the SBC-tree using the 3-sided structure or the R-tree achieves around 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 9, we present the relative performance of the SBC-tree insertion operation. The figure presents the average number of I/O operations performed by the SBC-tree to insert all RLE-suffixes relative to the average number of I/O operations performed by the String B-tree to insert all uncompressed 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 the SBC-trees index the RLE-suffixes that are significantly fewer than the suffixes of the uncompressed sequences. The big I/O saving achieved by the one-level SBC-tree is because of inserting the RLE-suffixes into the String B-tree, and no further I/Os are required. However, the SBC-tree using the 3-sided structure or the R-tree requires, in addition to inserting the RLE-suffixes into the String B-tree, inserting a point into the two-dimensional space for each inserted RLE-suffix.

In Figure 10, we present the SBC-tree I/O performance under *prefix matching* queries. The figure presents the average number of I/O operations performed by the SBC-tree relative to the average number of I/O operations performed by the String B-tree. 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 around

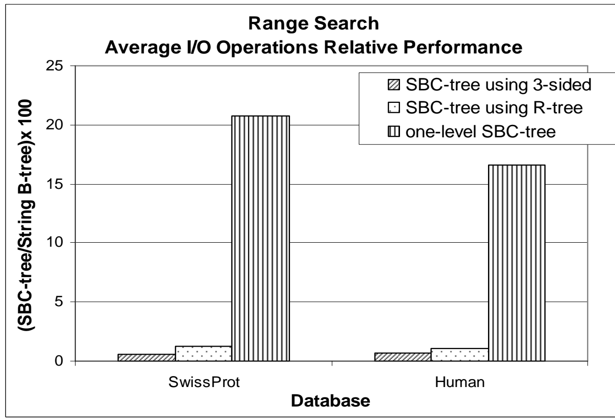


Figure 11: The performance under *range search* queries

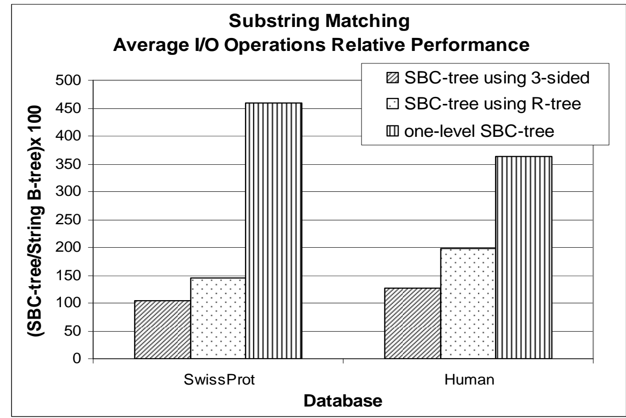


Figure 12: The performance under *substring matching* queries

85% I/O reduction. This I/O saving is because the SBC-tree searches a range of RLE-suffixes that is significantly smaller than the range of uncompressed suffixes that is searched by the String B-tree. The big difference between the performance of the one-level SBC-tree and the performance of the SBC-tree using either the 3-sided structure or the R-tree is because the size of the query answer relative to the size of the searched range, i.e., the range specified by *min_tag* and *max_tag*, is usually very small. The one-level SBC-tree scans the entire range to retrieve the query answer set, whereas the 3-sided structure and the R-tree retrieve only the query answer set from the specified range.

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 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 11. The figure illustrates that SBC-tree variants exhibit behavior similar to that of the *prefix matching* queries. The I/O saving in the case of *range search* queries is slightly less than that in the case of *prefix matching* queries because *range search* queries usually involve larger answer sets. 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 12. 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. Therefore, at least the same number of I/Os is performed by the SBC-tree to only retrieve the query answer set. The SBC-tree using the 3-sided structure is the best among the SBC-tree variants with 3% and 30% additional I/O overhead for the Swissprot and Human databases, respectively. 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. This range is larger than the range scanned by the String B-tree of the uncompressed sequences because we ignore the first RLE-character in the query pattern, which enlarges the searched range.

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 85% reduction in storage and 30% reduction in insertion I/Os.

8. CONCLUSION

We presented the SBC-tree index structure for indexing and searching RLE-compressed sequences of arbitrary length. The SBC-tree is a two-level index structure, the first level is the String B-tree and the second level is the 3-sided range query structure. The SBC-tree supports *substring matching*, *prefix matching*, and *range search* operations over RLE-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. We presented also two variants of the SBC-tree: the SBC-tree using the R-tree and the one-level SBC-tree, that do not have provable worst-case theoretical bounds for search operations, but perform well in practice. Our performance results illustrate that the SBC-tree using the 3-sided structure achieves up to 85% 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.

9. REFERENCES

[1] A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *DCC*, pages 279–288, 1992.

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

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

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

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

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

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

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

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

[10] R. Bayer and K. Unterauer. Prefix b-trees. *ACM Transactions on Database Systems*, 2(1):11–26, 1977.

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

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

[13] 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.

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

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

[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] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429, 1994.

[19] 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.

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

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

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

[23] 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.

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

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

[26] 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.

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

[28] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[29] L. Hammel and J. M. Patel. Searching on the secondary structure of protein sequences. In *VLDB*, pages 634–645, 2002.

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

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

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

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

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

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

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

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

[38] G. Navarro. Regular expression searching on compressed text. *Journal of Discrete Algorithms*, 1(5-6):423–443, 2003.

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

[40] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A model for sequence databases. In *ICDE*, pages 232–239, 1995.

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

[42] H. Tanaka and A. L. Garcia. Efficient run-length encodings. *IEEE Transactions 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. *Theoretical Computer Science*, 60(3):231–254, 1988.

[45] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE 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 *14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

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

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