WaveletTrie: a Compressed Self-Indexed Data Structure Supporting Efficient Database Operations

Stefan Böttcher, Rita Hartel, Jonas Manuel
Department of Computer Science,
University of Paderborn,
Paderborn, Germany
email: stb@uni-paderborn.de, rst@uni-paderborn.de, jmanuel@live.uni-paderborn.de

***Abstract*—An indexed compressed sequence of strings is a representation of a string sequence that on the one hand is more space efficient than the original sequence, but on the other hand supports more efficient operations on these strings. Whenever an indexed compressed sequence of strings supports efficient evaluation of typical database operations, like searching for exact matches or prefixes, range queries, computing the union or the intersection of sets of strings, or data modifications, databases can strongly benefit from storing their table columns in form of an indexed compressed sequence. In this paper, we show how to extend the data structure of the Wavelet Trie to an indexed compressed sequence of strings that supports efficient operations on column oriented databases. Therefore, we present algorithms for executing database operations like union, intersection, and range-queries on string sequences represented by an extended Wavelet Trie. Furthermore, in our evaluation, we show that performing these typical database operations on the extended Wavelet Trie is faster than simulating these operations on gzip- or bzip2-compressed data.**

***Keywords*—Column-oriented database management systems; compression; compressed indexed sequences of strings.**

I. INTRODUCTION

The work presented in this paper extends the ideas presented in [1] at DBKDA 2017.

Column-oriented DataBase Management Systems (DBMS) organize their data tables within column stores, each containing an ordered sequence of entries. This data organization technique is preferable, especially when used for read-intensive applications like data warehouses, where in order to analyze the data, queries and aggregates have to be evaluated on sequences of similar data contained in a single column [2]. A second advantage of column-oriented data stores is that they can be compressed stronger than row oriented data stores, as each column and therefore each contiguous sequence of data contains data from the same domain and thus contains less entropy.

As long as main-memory availability is a run-time bottleneck, data compression is beneficial to virtually "enhance" the capacity of the main-memory, i.e., column-oriented data stores can benefit from storing their string columns in form of *compressed indexed sequences of strings*. A major challenge when using a compressed data structure for a string column is to support typical database operations in efficient time without full decompression of the compressed data structure.

This paper is organized as follows. In Section III, we introduce the basic concepts used in the following sections. In Section IV, we explain different operations on the Wavelet Trie and discuss how to implement them. In Section V, we show an extensive performance evaluation, in which we compare the performance of these operations on the Wavelet Trie with the performance of the gzip and bzip2 (de)compression.

II. RELATED WORK

Column stores like, for instance, C-STORE [2], Vertica [3], or SAP HANA [4] typically rely on combinations of compression techniques like Run-Length Encoding, Delta Encoding, or dictionary-based approaches. These compression techniques do not contain a self-index, but have to occupy additional space to store an index that allows for efficient operations like, for instance the evaluation of range queries. When main-memory availability is the major run-time bottleneck, we consider this to be a disadvantage.

In contrast, the Wavelet Tree [5] is a self-index data structure, and it can be regarded as an enhancement of variable length encodings (e.g., Huffman [6], Hu-Tucker [7]). The Wavelet Tree rearranges the encoded string S in form of a tree and thereby allows for random access to S. Variations of the Wavelet Tree use the tree topology to enhance Fibonacci encoded data [8] or Elias and Rice variable length encoded data [9]. In [10] an n-ary Wavelet Tree is used instead of a binary Wavelet Tree (e.g., a 128-ary Wavelet Tree by using bytes instead of bits in each node of the Wavelet Tree). A pruned form of the Wavelet Tree is the Skeleton Huffman tree [11] leading to a more compressed representation. Although avoiding the need for an additional index, Wavelet Trees have the disadvantage that common prefixes in multiple strings are stored multiple times.

This disadvantage is avoided by the Wavelet Trie [12][13], which is a self-index, i.e., avoids the storage of extra index structures, and can be regarded as a generalization of the Wavelet Tree for string sequences S and the Patricia Trie [14]. The Wavelet Trie rearranges a sequence S of encoded strings $s_1,\ldots,s_n$ in form of a tree thereby storing common prefixes of $s_1,\ldots,s_n$ only once, and it allows for random access to each s_i of S. That is why in this paper, we

use a Wavelet Trie to store compressed indexed sequences of strings.

Wavelet Tries support the following basic operations that are used within column-oriented DBMS: the operations *access(n)* that returns the n-th string of a given column and that is used for example when finding values of the same database tuple contained in other columns, or *search(s)/searchPrefix(s)* that searches for all positions within the current column that contain a string value s (or that contain a string value having the prefix s). Beside these elementary search operations, Wavelet Tries support elementary data manipulation operations on the compressed data format as, e.g., to insert a string at a given position, to append a string, or to delete a string from the sequence.

[12] and [13] introduce the concept of the Wavelet Trie and discuss the complexity of the following operations, which are executed on the Wavelet Trie encoding a given string sequence:

- Access(pos) returns the pos-th string of the string sequence
- Rank(s, pos)/RankPrefix(s, pos) return the number of occurrences of string s (or strings starting with the prefix s) up to position pos in the string sequence
- Select(s, i)/SelectPrefix(s, i) returns the position of the i-th string s (or string starting with prefix s) of the string sequence
- Insert(s, pos) simulate on the Wavelet Trie an insertion of the string s before position pos into the string sequence
- Append(s) simulate on the Wavelet Trie an appending of the string s to the end of the string sequence
- Delete(pos) simulate on the Wavelet Trie a deletion of the string at position pos of the string sequence

Although this is sufficient to support the most elementary database operations in column stores, in order to support more enhanced data analysis, efficient query processing should go beyond these elementary operations. Hereby, the main challenge is to support efficient complex read operations like intersection, union, and range queries on column stores without decompression of large parts of the compressed data.

Our goal is that these operations on compressed data are executed not only with a smaller main-memory footprint, but also faster on compressed data compared to a decompress-read approach that first decompresses the data before a read operation (or write operation) is done.

Our first contribution is to extend the Wavelet Trie [12][13] published in 2012 by Grossi and Gupta by concepts and efficient implementations of enhanced database operations (intersection, union, and range queries).

When standard string compressors like gzip or bzip2 are used for compressing sequences of strings stored in a database table column, such a compressed sequence of strings has to be decompressed, before a database operation can even be executed. That is why our second contribution is an evaluation, comparing the performance of database operations on our extended Wavelet Trie with the decompression time needed by bzip2 or by gzip for a compressed sequence of strings. We show that performing typical operations on string sequences like searching for exact matches or prefixes, range queries, or update operations like insertion or deletion, or operations on two string sequences like merge or intersection, directly on the Wavelet Trie is faster than simulating these operations on bzip2- and gzip-compressed data.

III. Preprocessing and Basic Concepts

In this section, we introduce the basic concepts used in the remainder of this paper.

A. Preprocessing

In order to allow a space-efficient storage of a string sequence SS, the Wavelet Trie requires the strings of the string sequence to be pairwise prefix-free, i.e., no string $st_i \in SS$ is allowed to be a prefix of another string $st_j \in SS$. The requirement that SS has to be a prefix-free sequence, is not a critical restriction, as a prefix-free set of strings can be easily constructed for any set SS as follows. A special terminal symbol that is lexicographically smaller than each character occurring in each string $st_i \in SS$ is appended to each string $st_i \in SS$. For example, in Figure 1, we added the symbol '$' to the end of each string, thereby yielding a prefix free sequence, although the previously given string sequence (rob, romulus, robert) is not prefix free, as 'rob' is a prefix of 'robert'.

Furthermore, before we use the Wavelet Trie to store a sequence $SS=(st_1,\ldots,st_n)$ of strings, we apply the Hu-Tucker algorithm [7] to all characters of all $st_i \in SS$, i.e., we encode each character c of each $st_i \in SS$ by its Hu-Tucker code ht(c). Thereby, we get a sequence $S=(s_1,\ldots,s_n)$ of bit-strings, where s_i is the Hu-Tucker code of st_i, yielding a lexicographic Huffman code for the bit-strings s_i, i.e., $s_i <_{lexi} s_j \Leftrightarrow st_i <_{lexi} st_j$, where $<_{lexi}$ denotes "less than in lexicographical order".

In order to search for a string st in SS, we apply the same Hu-Tucker codes to the characters of st, yielding a bit-string s that we can search for in the Wavelet Trie.

For example, for the string sequence SS=(rob$, romulus$, robert$), and its Hu-Tucker codes listed in Table 1, we get the sequence
$S=(s_1, s_2, s_3)$ of bit-strings with
s_1 = 101 100 0100 00,
s_2 = 101 100 0111 111 0110 111 1100 00, and
s_3 = 101 100 0100 0101 101 1101 00.

TABLE I. Hu-Tucker codes of our examples

char c	code(c)	**char c**	code(c)
$	00	**o**	100
b	0100	**r**	101
e	0101	**s**	1100
l	0110	**t**	1101
m	0111	**u**	111

Whenever we describe operations that work on two Wavelet Tries, we assume that both Wavelet Tries are based on the same Hu-Tucker-Encoding.

After the preprocessing steps, we construct the Wavelet Trie according to the construction method given in the Wavelet Trie definition of the following section.

In order to keep the following definition and the construction principle of the Wavelet Trie simple, we assume that the string sequence SS represented by the Wavelet Trie always consists of at least two different strings, i.e., $st_i \in SS$, $st_j \in SS$, and $st_i \neq st_i$. As a consequence, also SS's Hu-Tucker encoded sequence $S=(s_1,\ldots,s_n)$ of bit-strings contains at least two different bit-strings. Whenever this is not the case, we could simply store the size of S externally, or add a binary string s_{new}, $s_{new} \neq s_i$, $i \in \{1,\ldots,n\}$ to the end of the sequence S, such that our Wavelet Trie represents a bit-string sequence S that consists of at least two different bit-strings.

B. Basic Concepts

Similarly to [12][13], we define the Wavelet Trie as follows:

Definition (Wavelet Trie): Let S be a non-empty sequence of bit-strings, $S=(s_1,\ldots, s_n)$, $s_i \in \{0,1\}^*$, whose underlying bit-string set $S_{set}=\{s_1,\ldots,s_n\}$ is pair-wise prefix-free. The Wavelet Trie of S, denoted WT(S), is a binary tree, which is built recursively as follows:

(i) If the bit-string set S_{set} of S consists of a single bit-string element only, i.e., $s_1 = \ldots = s_n$, the Wavelet Trie is a single node labeled with $\alpha = s_1 = \ldots = s_n$ and $\beta=\varepsilon$, i.e., β is an empty bit-vector.

(ii) Otherwise, WT(S), the Wavelet Trie of S is a binary tree whose root node is labeled with two bit-vectors, α and β, and whose children (respectively labeled with 0 and 1) are $WT(S_{\alpha 0})$ and $WT(S_{\alpha 1})$, the Wavelet Tries of sequences $S_{\alpha 0}$ and $S_{\alpha 1}$ of bit-strings, where α, β, $S_{\alpha 0}$, and $S_{\alpha 1}$ are defined as follows. α is the longest common prefix of the bit-strings s_i contained in S. For any $1 \leq i \leq n$, we then can write $s_i = \alpha b_i \gamma_i$, where b_i is a single bit. We then define the sequences $S_{\alpha 0} = (\gamma_i \mid \alpha 0\gamma_i \in S)$ and $S_{\alpha 1} = (\gamma_i \mid \alpha 1\gamma_i \in S)$ of bit-strings, i.e., the first sequence contains suffixes γ_i of $s_i \in S$ such that s_i begins with $\alpha 0$ and the second sequence contains suffixes γ_i of $s_i \in S$ such that s_i begins with $\alpha 1$. Finally, the bitvector $\beta = (b_i \mid \alpha b_i \gamma_i \in S)$ collects all the bits b_i that discriminate for a given $s_i = \alpha b_i \gamma_i$, whether the suffix γ_i is in $S_{\alpha 0}$ or is in $S_{\alpha 1}$.

Example: Continuing the previous example, Figure 1 shows the Wavelet Trie representing the sequence $S=(s_1, s_2, s_3)$ of bit-strings

s_1 = 101 100 01 0 0 0 0,
s_2 = 101 100 01 1 1 111 0110 111 1100 00,
s_3 = 101 100 01 0 0 0 1 01 101 1101 00.

representing the strings of the sequence SS= (rob$, romulus$, robert$). The characters are shown in Figure 1 only for the purpose of illustration, but are not contained in the Wavelet Trie. All three bit-strings s_1, s_2, s_3 share the common prefix α = 101 100 01 and differ in the next bit b_i, which is 0 for s_1 and for s_3, but 1 for s_2. The bit-string sequence $S_{\alpha 0} = (\gamma_1, \gamma_3)$ containing the suffixes γ_1 and γ_3 of s_1 and s_3 is represented by the left sub-tree of the Wavelet Trie's root node, and the bit-string sequence $S_{\alpha 1} = (\gamma_2)$ is represented by the right child. Continuing with the left sub-tree of the Wavelet Trie's root node, γ_1 and γ_3 share the common prefix 00 and differ in the next bit, such that here we get a new bit-vector α=00 and a new bit-vector β that contains only two bits, one for s_1, the other for s_3. All leaf nodes contain the common bit-vector α and an empty bit-vector β, as they represent a bit-string set containing a single element only.

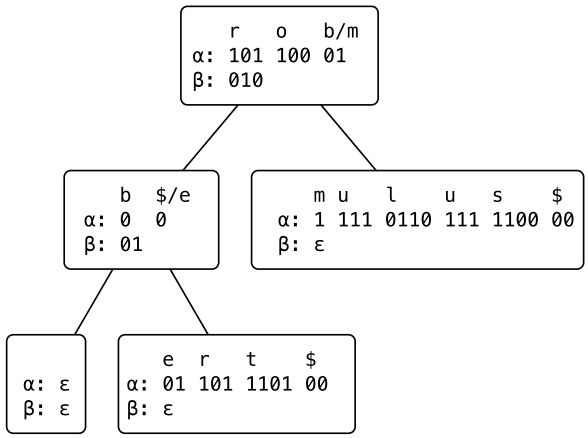

Figure 1. Wavelet Trie representing the bit-string sequence (101 100 01 0 0 0 0, 101 100 01 1 1 111 0110 111 1100 00, 101 100 01 0 0 0 1 01 101 1101 00)

Remarks: If the whole Wavelet Trie consisted of a single node only, this node would be a leaf node having a bit-vector α, but having an empty bit-vector β, i.e., we could not derive the size of the sequence S represented by the leaf node. That is why we required S to contain at least two different bit-strings $s_i, s_j \in S$ with $s_i \neq s_j$. Given this requirement, in the following, we can assume that the whole Wavelet Trie considered always consists of more than one node.

If we reach a leaf node of the Wavelet Trie, i.e., the sequence represented by that leaf node has a bit-string set containing a single element only, we know the size n of the sequence by counting the number of bits within the β bit-vector of the leaf's parent node.

IV. Database Operations on top of the Wavelet Trie

In the following sections, we denote the left child of a Wavelet Trie node also as its 0-child, and the right child of a Wavelet Trie node also as its 1-child.

For the following operations, we assume that they are methods of an object of type WaveletTrie and that they can directly access the object variables α, β, and size. The size of a Wavelet Trie wt is $|\beta|$, i.e., the number of bits of β if wt is an inner node, and for a leaf node, it is the number of 0-bits in wt's parent node p, if wt is p's left child, and the

number of 1-bits in wt's parent node p, if wt is p's right child.

A. Search/Query Operations

Let S=(s_1,..., s_n) be a given bit-string sequence that is represented by a Wavelet Trie wt having the root node wt.root, and let s be a given bit-string that we search for in S. The *search(s)* operation applied to wt.root returns a list of all the positions i in S of those bit-strings s_i that are equal to s. Similarly, the *searchPrefix(s)* operation applied to wt.root returns a list of the positions i in S of all those bit-strings s_i that have a prefix s.

For example, assume, we want to search for robert$ in the string sequence SS_2=(rob$, romulus$, robert$, robert$, romulus$, robert$), and we use the same Hu-Tucker coding as before. We search for the bit-string 101 100 0100 0101 101 1101 00 corresponding to robert$ in the Wavelet Trie corresponding to SS_2 (c.f. Figure 3), and we get the result position list [3,4,6].

In order to search for all matching positions in the Wavelet Trie, the function *search(s)* (c.f. Code 1) recursively traverses down the tree representing the Wavelet Trie, until we have found all bits of the binary string s, and bottom-up "translates" all matching positions into matching positions in the Wavelet Trie's root node.

In more detail, the search works as follows: Let n be the current node having a bit-vector α, a bit-vector β, and (if n is not the Wavelet Trie's root node) having a parent node pa, such that n is the b-child of pa and β of pa contains k b-bits. Assume that we search for all positions where a given bit-string s occurs.

```
1 Function search(BitString s):
2 if s=α and isLeaf then
3    return [1,...,size];
4 else if α.isPrefixOf(s) and !isLeaf then
5    Bit b = α.getFirstDifferentBit(s);
6    γ = s.getSuffix(α);
7    [p1,...,pm] = getChild(b).search(γ);
8    return β.select(b, [p1,...,pm]);
9 return ∅;
```

Code 1. Search for all exact occurrences of a word

The search has to distinguish 3 cases:

1. If we have found all bits of s and we have reached the end of the α of a leaf node (line 2), we have reached an exact match. For the leaf node representing the exact match, we return a vector [1,…,size] (line 3), where size is the number of bit-strings represented by that leaf, i.e., each position in [1,…,size] is a matching position for s in this leaf node.

2. If s≠α, α is a prefix of s and n is no leaf node (lines 4-8), s is of the form s=αbγ (lines 5-6) with a potentially empty γ. In this case, we perform the search operation for the bit-string γ on n's b-child (c.f. Figure 2), resulting in a list of matching positions [p_1,…,p_m] in the current node's b-child (c.f. line 7). These positions are then (line 8) translated into matching positions of the current node as described below.

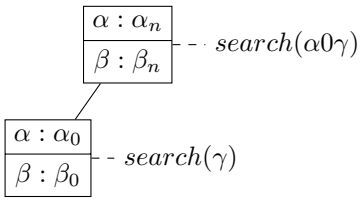

Figure 2. Search operation, if α is a prefix of search string s.

3. If none of the above cases matches, the Wavelet Trie does not contain a binary string matching the search criteria, and we return an empty list of positions (line 9).

In order to determine the exact matching positions in S, matching positions in the Wavelet Trie's nodes are computed bottom-up, i.e., we start in the matching leaf node and "translate" matching positions in a Wavelet Trie node n to matching positions in n's parent node pa (line 8).

Hereby, the method β.select(b, [p_1,…, p_m]), called in line 8 of Code 1, is implemented as shown in Code 2, where β.select(b,p_i) returns the position of the p_i^{th} b-bit in the bit-vector β.

```
1 Function select(Bit b, [p1,...,pm]):
2    return [select(b,p1),...,select(b,pm)];
```

Code 2. Auxiliary method β.select(b, [p_1,…, p_m])

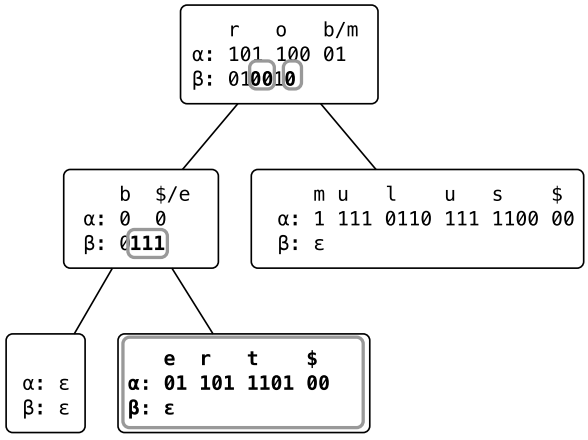

Figure 3. Search for "robert$".

Example: Figure 3 shows a slightly modified version of the Wavelet Trie of Figure 1 that represents the string sequence SS_2=(rob$, romulus$, robert$, robert$, romulus$, robert$). In order to search for all occurrences of "robert$", i.e., of the binary string s = 101 100 01 0 0 0101 101 1101 00, we start with the root node. As α = 101 100 01 is a prefix of s, and the next bit b in s is 0, we continue to search for the remaining substring s_L = 0 0101 101 1101 00 of s within the left child of the root node. Here, α = 0 0 is again a prefix

of s1, and the next bit b of s_L is 1. Therefore, we continue to search for the remaining substring s_{LR} = 01 101 1101 00 of s_L in the right child node. As now, $\alpha = s_{LR}$, and the current node is a leaf node, all strings represented by this leaf node are matches. From the leaf's parent's ß-bit-vector, we know that our leaf node represents 3 occurrences of s, thus, we return the list of positions [1, 2, 3] from the leaf node to its parent. Recursive execution returns to the previous call of this method executed for the leaf' parent, i.e., the left child of the root. As the leaf is the right child of its parent and here, the 1st, 2nd, and 3rd 1-bits occur at positions 2, 3, and 4 in β, we return [2, 3, 4]. Similarly, when execution returns from its left child to the root node, the 2nd, 3rd, and 4th 0-bits occur at positions 3,4, and 6 in β. Therefore, the final result is that the strings on positions [3, 4, 6] are equal to the search string 'robert$'.

Similarly, if we do not want to search for exact matches, but for prefixes, i.e., we search for all positions of binary strings starting with the given prefix s, we just have to slightly modify our algorithm:

If we have found all bits of s on the current path in the Wavelet Trie, we do not care, whether or not we have reached a leaf node, and translate the current positions into positions of the Wavelet Trie's root node. That is, the code for searchPrefix() is as follows:

```
1 Function searchPrefix(BitString s):
2 if s=α or s.isPrefixOf(α) then
3    return [1,...,size];
4 else if α.isPrefixOf(s) and !isLeaf then
5    Bit b = α.getFirstDifferentBit(s);
6    γ = s.getSuffix(α);
7    [p1,...,pm] = getChild(b).search(γ);
8    return β.select(b, [p1,...,pm]);
9 return ø;
```

Code 3. Search for all strings that start with a given prefix

B. *Between/Range Queries (less than, greater than)*

The operation between(s_L,s_H) returns a list of positions of strings $s_i \in S$, such that $s_L \leq_{lexi} s_i \leq_{lexi} s_H$. Similarly, the operation lessThan(s) returns a list of positions of bit-strings s_i, such that $s_i \leq_{lexi} s$, and the operation greaterThan(s) returns a list of positions of bit-strings s_i, such that $s_i \geq_{lexi} s$.

For example, assume, we want to search for all strings greater than robb$ in the string sequence SS_2=(rob$, romulus$, robert$, robert$, romulus$, robert$), and we use the same Hu-Tucker coding as before. We search for all bit-strings that are greater than the bit-string 101 100 0100 0100 00 corresponding to robb$ in the Wavelet Trie corresponding to SS_2 (c.f. Figure 4), and we get the result position list [2,5,3,4,6] corresponding to the positions of either one of the strings robert$ or one of the strings romulus$.

In this section, we explain how to implement the operation greaterThan(s). To adapt this operation in order to implement between(s_L,s_H) or lessThan(s) is quite straightforward.

Let the current node n be a node with labels α and β.

```
1 Function greaterThan(BitVector s):
2 if α>=s then
3   return [1...size];
4 else if α.isPrefixOf(s) then
5   if not isLeaf then
6     Bit b = α.getFirstDifferentBit(b);
7     λ = s.getSuffix(α);
8     if b=1 then
9       [p1,...,pm] = getChild(1)
                        .greaterThan(λ));
10      return β.select(1, [p1,...,pm]);
11    else
12      [p1,...,pm] = getChild(0)
                        .greaterThan(λ));
13      return
14      β.select(0, [p1,...,pm]) ⊕
15     β.select(1, [1,...,rank(1,|β|)]);
16 else return ø;
```

Code 4. Search for positions of all words greater than a given string

If α≥s (which includes the case that s is a prefix of α, i.e., α=sbδ with a potentially empty δ), we know that all bit-strings represented by the Wavelet Trie rooted in n are greater than or equal to s, i.e., we return the list of positions [1, …, size] (lines 2-3).

In the other case, if α is a prefix of s, i.e., s= αbλ (lines 4-16), we have to consider the value of b and get two sub-cases.

Lines 8-10 treat the sub-case b=1. Here, all remaining bit-strings represented by the Wavelet Trie rooted in n's 0-child start with a 0-bit and therefore cannot be greater than the remaining search bit-string 1λ of s, i.e., only n's 1-child can represent greater remaining bit-strings. As the 1-bit of the remaining search bit-string 1λ is consumed while search moves to n's 1-child, we have to apply the operation greaterThan(λ) only to n's 1-child (line 9) with the remaining search bit-string λ, and translate the matching positions [p_1,...,p_m] within n's 1-child's β-vector into matching positions wihtin n's β-vector (line 10).

Lines 11-15 treat the other sub-case, b=0:

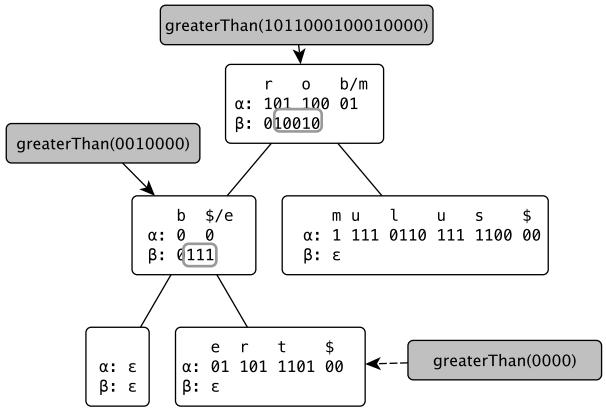

Figure 4. Search for strings greater than "robb$"

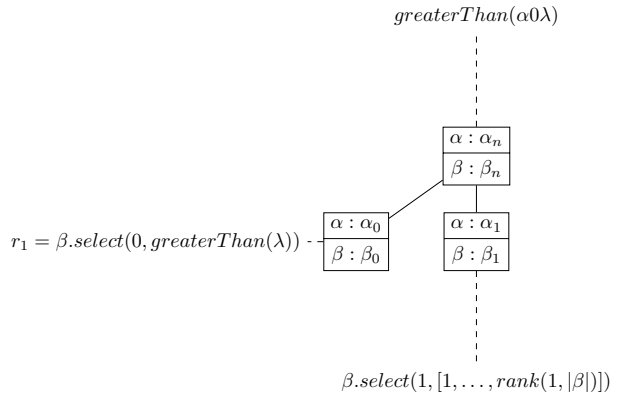

Figure 5. greaterThan if α is a prefix of search string s.

Here, the returned result list $r_1 \oplus r_2$ of matching positions wihtin n's β-vector consists of two sub-lists r_1 (line 14) and r_2 (line 15), which are computed as follows. We search for the positions $[p_1,...,p_m]$ of remaining bit-strings greater than λ within n's 0-child (line 12) and translate these positions into positions of n's β bit-vector (line 14) to get r_1 However, as all remaining bit-strings represented by the Wavelet Trie rooted in n's 1-child start with a 1-bit and are therefore greater than the remaining search bit-string 0λ of s, each 1-bit of n's β bit-vector represents a match to our search. That is why we also return the positions of all 1-bits in β in line 15 of Code 4, i.e., r_2 = β.select(1,[1,…,rank(1, |β|]). This is also illustrated in Figure 5. Finally, we return $r=r_1 \oplus r_2$ (lines 13-15).

In the last case, α<s and α is not a prefix of s, no string represented by this Wavelet Trie rooted in n can be greater than or equal to s, i.e., we return the empty list (line 16).

Example: Figure 4 shows the search for all strings greater than the binary string s = 101 100 0100 0100 00 (="robb$") within the string sequence SS_2=(rob$, romulus$, robert$, robert$, romulus$, robert$). We compare s with α of the Wavelet Trie's root. As α is a prefix of s and the next bit after α in s is 0, we select all 1-bits of β (positions 2 and 5) and compute greaterThan applied to the remaining bit-string s_L = 0 0100 00 and the left child of the root. Again α is a prefix of s_L, but this time, the next bit after α in s is 1, so we continue with the right child and the remaining bit-string s_{LR} = 0000 and compute greaterThan(0000). As $\alpha > s_{LR}$ for s_{LR} = 0000, we return positions [1, 2, 3], which correspond to the 1st, 2nd, and 3rd 1-bit of the β bit-vector of its parent, i.e., to the positions [2, 3, 4]. The positions [2, 3, 4] correspond to the 2nd, 3rd, and 4th 0-bit of the β bit-vector of the root, i.e., to positions [3, 4, 6]. So the final result are the positions [2, 5] ⊕ [3, 4, 6] = [2, 5, 3, 4, 6].

Note that we can similarly create a new Wavelet Trie that consists of all bit-strings greater than s, if we first copy the Wavelet Trie, and then, using the list $[p_1,\ldots, p_m]$ of result positions returned by greaterThan(s), for deleting all strings at a position pi ∈ $[p_1,\ldots, p_m]$ from the copy of the Wavelet Trie. Note that the algorithm for deleting a string at a given position is explained in the following section, Section C. Of course, we do not need to search first and delete in a second pass, but can nest both operations by deleting the bits while propagating the positions up to the root node.

C. Delete

This operation deletes the binary string s_{pos} at position pos from the Wavelet Trie.

For example, assume, we want to delete the 3rd string from string sequence SS=(rob$, romulus$, robert$), and we use the same Hu-Tucker coding as before. We delete the binary string 101 100 0100 0101 101 1101 00 corresponding to robert$ from the Wavelet Trie corresponding to SS (c.f. Figure 7). As a consequence, we have to *collapse* the left child of the root, i.e., to combine it with its remaining single child into a single node.

```
1 Function delete (int pos):
2 if not isLeaf then
3    boolean b = β.getBit(pos);
4    int rank = β.delete(pos);
5    if β.hasOnly(1-b) then
6       collapse(1-b);
7    else
8       getChild(b).delete(rank);
```

Code 5. Delete word from WaveletTrie

In order to delete the binary string s_{pos} at position pos from the current node n (initially the root node), we delete the bit b at position pos from β (Code 5, line 4). If β afterwards still contains 0-bits and 1-bits and n's b-child is not a leaf node, we continue to delete the bit at position rank(b,pos) from the β-vector of n's b-child (lines 7-8).

If β contains either only 0-bits or only 1-bits (i.e., β=0* or β=1*, in general β=b*), we have to collapse the current node with its b-child n_b and thereby delete its (1-b)-child (lines 5-6).

Figure 6 shows the state before and after collapsing the node for b=0. To collapse a node n with its b-child n_b means the following: Let n_0 be the 0-child of n_b and n_1 be the 1-child of n_b. Let furthermore α_b and β_b be the labels of node n_b. Then the new labels of node n are $\alpha=\alpha_n b \alpha_b$ and $\beta=\beta_b$. Furthermore, n_0 becomes the new 0-child of n and n_1 becomes the new 1-child of n.

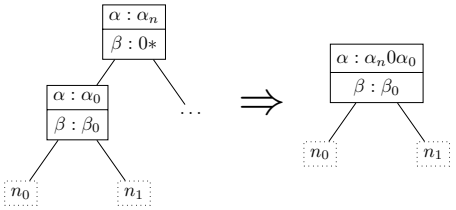

Figure 6. Before and after collapsing the node.

Example: Figure 7 shows the process of deleting the string s=101 100 0100 0101 101 1101 00 (=robert$) at position 3 from the WaveletTrie. In the root, we delete the 3rd bit, which is a 0-bit. We compute its rank within β. As it is the second 0-bit, the rank is two. We continue with deleting

the second bit of the left child. As it is the single 1-bit, we can delete the right child. As the node now only has a left child and no right child, we finally collapse it with its left child.

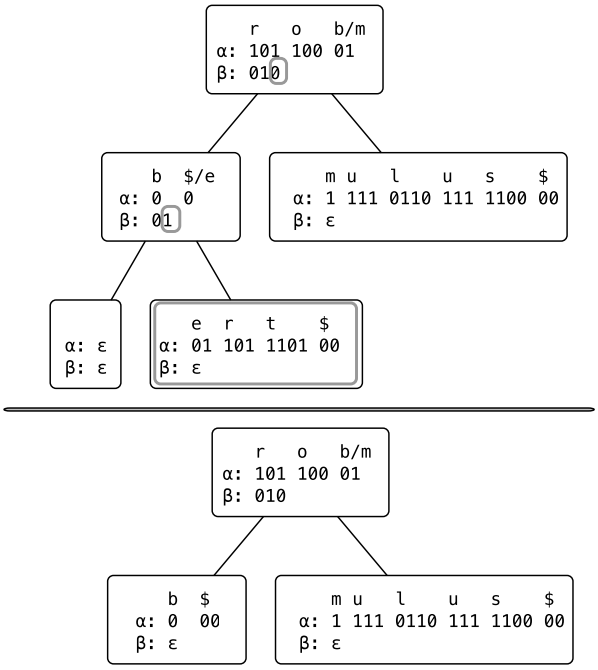

Figure 7. Deleting the 3rd string

D. Insert/Append

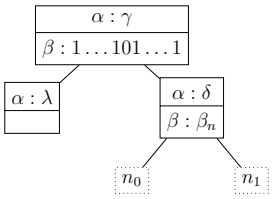

Figure 8. Result of insert operation if s and α_n have a common prefix.

This operation inserts a binary string s into the Wavelet Trie representing a string sequence S=(s_1,…,s_n) at position pos (or appends it to the end, if pos>n). Note that this operation is only defined if the set $S_{set} \cup \{s\}$ is prefix free.

For example, assume, we want to insert the string romulus$ (bit-string s=101 100 01 1 1 111 0110 111 1100 00) at position 2 into the string sequence SS_3=(rob$, robert$), and we use the same Hu-Tucker coding as before. We insert s into the Wavelet Trie corresponding to SS_3 (c.f. Figure 9), thereby splitting the previous root into the new root representing the common binary sub-string 101 100 01 of all three binary strings and a new 0-child representing the remaining common binary sub-string 00 of the binary strings of SS_3.

```
1  Function insert (int pos, BitVector s):
2  if αn = s then
3      return;
4  if αn.isPrefixOf(s) then
5      boolean b =αn.getFirstDifferentBit(s);
6      int rank = βn.insert(pos, b);
7      δ = s.getSuffix(αn);
8      getChild(b).insert(rank, δ);
9  else
10     BitVector γ = αn.getCommonPrefix(s);
11     boolean b = γ.getFirstDifferentBit(s);
12     BitVector δ  = αn.getSuffix(γ);
13     BitVector λ = s.getSuffix(γ);
14     α = γ;
15     BitVector β =BitVector(1-b, |βn|);
16     β.insert(pos, b);
17     setChild(b, WaveletTrie(λ));
18     setChild(b, WaveletTrie(δ, βn,
19                     child 0 , child 1 ));
```

Code 6. Append or insert a word to WaveletTrie at position pos

Code 6 summarizes the implementation of the insert operation. We consider the current node n of the Wavelet Trie (initially the root node) having the labels α_n and β_n and the binary string s to be inserted at position pos. As we require the Wavelet Trie before and after the insertion to be prefix free, we know that s must not be a prefix of α_n.

If α_n=s (lines 2-3), n must be a leaf node of the Wavelet Trie (as otherwise the Wavelet Trie would not be prefix free after the insertion). As the size of the leaf node is stored in the β-bit-vector of its parent, the insertion is completed and we do not need to do anything else.

If α_n is a prefix of s (lines 4-8), i.e., s=α_nbδ, where b is a bit, we insert bit b at position pos into β_n(line 6) and remember its rank, and insert the binary string δ into the b-child of n at the position remembered in that rank, i.e., rank(b,pos) (line 8).

Otherwise (lines 9-19), let γ be the common prefix of α_n and s (line 10) and let us assume w.l.o.g. that α_n=γ1δ (line 12) and s=γ0λ (line 13). This scenario is illustrated in Figure 8. Note that γ might even be an empty binary string. Let n_0 be the 0-child of n, and let n_1 be the 1-child of the current node. In this case, we change n into a node with α=γ (line 14), and β consists of $|\beta_n|$ 1-bits (line 15) and one 0-bit at position pos (line 16). The new 0-child of n is a node n' with α=λ (line 17). The new 1-child of n is a node n'' with α=δ and β=β_n(line 18). n'' gets n_0 as 0-child and n_1 as 1-child (line 19). Figure 8 shows this case after having inserted s into α_n.

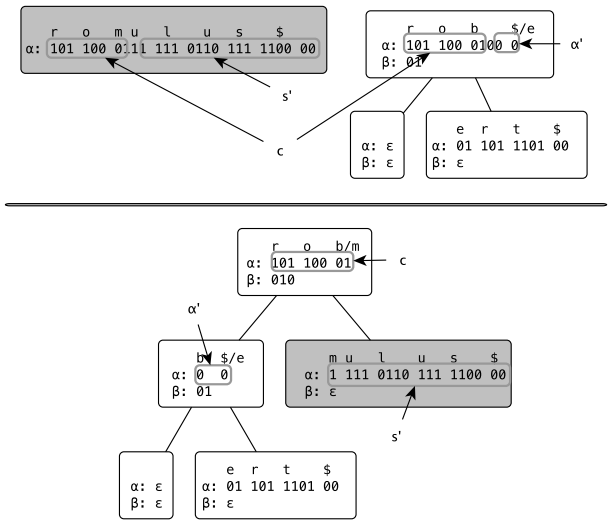

Figure 9. Inserting "romulus$" into the WaveletTrie

Example: Figure 9 shows the process of inserting the string s=101 100 0111 111 0110 111 1100 00 (=romulus$) as second string into the Wavelet Trie representing the string sequence SS_3=(rob$, robert$). We split s into the common prefix of s and α c=101 100 01, the next bit b=1, and the suffix s'=1 111 0110 111 1100 00 . And we split α into c, the next bit b'=0 and the suffix α'=00. The α-bit-vector of the new root is α=c, β consists of 2 bits b'=0 representing the existing Wavelet Trie, in which we insert the bit b=1 at position 2. The left child of the new root becomes a node with α= α' and the beta of the left child is a copy of β=01. Furthermore, the two children of the left child are copies of the two child nodes of the previous root of the existing Wavelet Trie. The right child of the new root node becomes a new node with α=s' and β=ε.

E. Merge/Append + Union

This operation simulates the insertion of one string sequence SS_2 at a given position pos into another string sequence SS_1 on two Wavelet Tries t_1 representing SS_1 and t_2 representing SS_2. Note that this operation is only defined if the set $s_1 \cup s_2$ is prefix free.

For example, if we merge a copy of the Wavelet Trie in Figure 1 representing the string sequence SS= (rob$, romulus$, robert$) at position 2 into another copy of that Wavelet Trie, we get a new Wavelet Trie that represents the string sequence (rob$, romulus$, rob$, romulus$, robert$, robert$)

Let n_1 be the current node of t_1, i.e., the given Wavelet Trie, and n_2 be the current node of t_2, i.e., the Wavelet Trie to be merged or appended, where n_1 has the labels α_1 and β_1 and n_2 has the labels α_2 and β_2.

If $\alpha_1=\alpha_2$ and n_1 and n_2 are leaf nodes, nothing has to be done, and the operation is finished (lines 3-4).

Note that if $\alpha_1=\alpha_2$, the cases that n_1 is a leaf node, but n_2 is not a leaf node, and vice versa, cannot occur for the following reason. If $\alpha_1=\alpha_2$ and at least one node, n_1 or n_2 is not a leaf, also the other node must not be a leaf (line 5), because otherwise, $s_1 \cup s_2$ would not be prefix-free. Then, we insert β_2 at position pos into β_1 (line 6), merge the 0-child of n_2 at position β_1.rank(0,pos) into the 0-child of n_1 (line 7) and merge the 1-child of n_2 at position β_1.rank(1,pos) into the 1-child of n_1 (line 8).

```
 1 Function merge(int pos, WaveletTrie t):
 2 if t.α= α then
 3   if isLeaf and t.isLeaf then
 4     return;
 5   /*neither the current node nor t
                  is a leaf node */
 6   Rank r = β.insert(pos, t.β);
 7   getChild(0).merge(r(0),t.getChild(0);
 8   getChild(1).merge(r(1),t.getChild(1);
 9 if α.isPrefixOf(t.α) then
10   Bit b = α.getFirstDifferentBit(t.α);
11   for i in 1..|t.β| do
12     Rank r = β.insert(pos, b);
13   t.α = t.α.getSuffix(α);
14   getChild(b).merge(r(b),t);
15 else if t.α.isPrefixOf(α) then
16   Bit b = t.α.getFirstDifferentBit(α);
17   WaveletTrie tmp = WaveletTrie
        (α.getSuffix(t.α), β,getChildren());
18   α = t.α;
19   β = BitVector(b, |tmp|);
20   Rank r = β.insert(pos, t.β);
21   tmp.merge(r(b),t.getChild(b));
22   setChild(b,tmp);
23   setChild(1-b,t.getChild(1-b));
24 else
25   BitVector γ = α.getCommonPrefix(t.α);
26   Bit b = γ.getFirstDifferentBit(t.α);
27   WaveletTrie t1 = WaveletTrie
    (t.α.getSuffix(γ),t.β,t.getChildren());
28   WaveletTrie t2 = WaveletTrie
        (t.α.getSuffix(γ),β,getChildren());
29   α = γ;
30   β = BitVector(1-b, |β|);
31   for i in 1..|t.β| do
32     β.insert(pos, b);
33   setChild(b,t1);
34   setChild(1-b,t2);
```

Code 7. Merge two Wavelet Tries

If α_1 is a prefix of α_2, i.e., $\alpha_2=\alpha_1 b\delta$ (lines 9-14), we insert b $|\beta_2|$ times at position pos into β_1 (lines 10-12). We change α_2 into δ (line 13) and merge n_2 into the b-child of n_1 at position β_1.rank(b,pos) (line 14).

If α_2 is a prefix of α_1, i.e., $\alpha_1=\alpha_2 b\lambda$ (lines 15-23), we create a new node n with labels $\alpha=\alpha_2$ (line 18) and β consisting of $|\beta_1|$ bits b, in which we insert β_2 at position pos (lines 19-20). The b-child of the node n is then the result of merging the b-child of n_2 at position β_1.rank(b,pos) into a node with labels $\alpha=\gamma$ and $\beta=\beta_1$, having the children of n_1 as children (lines 17, 22). The (1-b)-child of the node n is the (1-b)-child of n_2 (line 23).

Otherwise, α_1 and α_2 share a common prefix (lines 25-34). Let γ be the common prefix of α_1 and α_2, and let us assume w.l.o.g. that $\alpha_1=\gamma 1\delta$ and $\alpha_2=\gamma 0\lambda$ (c.f. Figure 10). Then, we modify the current node n by setting the label $\alpha=\gamma$ (line 29) and β to a bit-vector consisting of $|\beta_1|$ bits 1, in which we insert $|\beta_2|$ bits 0 at position pos (lines 30-32). The 0-child of node n is then a node with $\alpha=\lambda$ and $\beta=\beta_2$ having the children of n_2 as child nodes (lines 28,34), and the 1-child of node n is a node with $\alpha=\delta$ and $\beta=\beta_1$ having the children of n_1 as child nodes (lines 27,33).

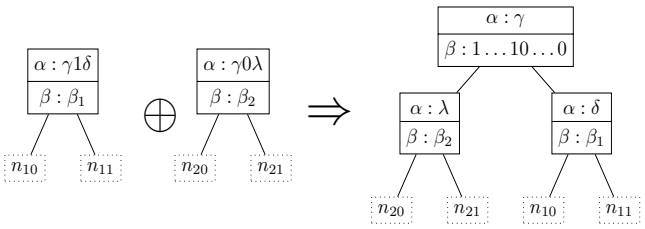

Figure 10. Appending t_1 and t_2 if α_1 and α_2 share a common prefix.

F. Intersection

This operation (c.f. Code 8) computes the set-intersection of two Wavelet Tries t_1 and t_2, i.e., it computes a Wavelet Trie that represents a lexicographically ordered list of all bit-strings s that occur in both, t_1 and in t_2.

Let n_1 be the current node of t_1 and n_2 be the current node of t_2, where n_1 has the labels α_1 and β_1 and n_2 has the labels α_2 and β_2.

If $\alpha_1=\alpha_2$ and n_1 and n_2 are leaf nodes, we can return n_1 as resulting Wavelet Trie (lines 3-4), as n_1 represents the intersection.

If $\alpha_1=\alpha_2$ and n_1 and n_2 are inner nodes, we compute the result node r_0 of the intersection of the 0-child of n_1 and of the 0-child of n_2 (line 6) and the result node r_1 of the intersection of the 1-child of n_1 and of the 1-child of n_2 (line 7). If both, the intersection of the 0-children of n_1 and n_2 and the intersection of the 1-children of n_1 and n_2 are empty (line 8), we return that the intersection of n_1 and n_2 is empty. Otherwise (lines 10-14), we return a new node n, with $\alpha=\alpha_1$, β consists of $|r_0|$ 0 bits followed by $|r_1|$ 1 bits, and n has r_0 as 0-child and has r_1 as 1-child. Note that by using this order of 0-bits and 1-bits in n's β-vector, we get the intersection in lexicographical order. If only one of the intersections is empty, we collapse the new node with the child representing the non-empty intersection (lines 10-11).

Let now either α_1 be a prefix of α_2 (lines 15-16) or vice versa (lines 17-18). Let us assume w.l.o.g. that α_2 is a prefix of α_1, i.e., $\alpha_1=\alpha_2 b\gamma$ (c.f. Figure 11). In this case, we change α_1 into $\alpha_1=\gamma$ and intersect this new node with the b-child of n_2 (line 4a). If the intersection does not return an empty node (line 5a), the β of the result node contains only bits b (i.e., only 1 bits or only 0 bits) as it does not have a (1-b)-child. Therefore, we collapse it with its single child node b-child n_b and thereby delete its (1-b)-child (lines 7a-10a).

```
 1 Function intersection(WaveletTrie t1,
                         WaveletTrie t2):
 2 if t1.α= t2.α then
 3   if t1.isLeaf and t2.isLeaf then
 4     return t1
 5   else if not t1.isLeaf and
           not t2.isLeaf then
 6     WaveletTrie r0 = intersection
        (t1.getChild(0), t2.getChild(0));
 7     WaveletTrie r1 = intersection
         (t1.getChild(1),t2.getChild(1));
 8     if r1=ø and r2=ø then return ø;
 9     else
10       res = WaveletTrie(t1.α,-,r0,r1);
11       if r0 = ø then res.collapse(1);
12       if r1 = ø then res.collapse(0);
13       res.β  = 1^|r0| 0^|r1|;
14       return res;
15 if t1.α isPrefixOf t2.α then
16   return intersectionPrefix(t1,t2);
17 else if t2.α isPrefixOf t1.α then
18   return intersectionPrefix(t2,t1);
19 else return ø;

1a Function intersectionPrefix(
     WaveletTrie pre, WaveletTrie other):
2a Bit b = pre.α.getFirstDifferentBit
                                (other.α);
3a other.α = other.α.getSuffix(pre.α);
4a WaveletTrie res_b =
     intersection(pre.getChild(b),other);
5a if res_b = ø then return ø;
6a else
7a   res.α = pre.α;
8a   res.setChild(b,res_b);
9a   res.collaps(b);
10a  return res;
```

Code 8. Compute the intersection of two Wavelet Tries

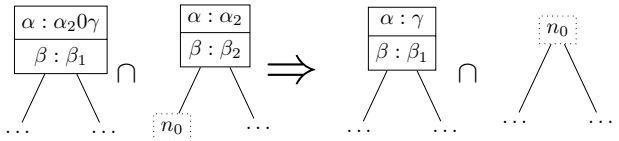

Figure 11. Intersection if α_2 is a prefix of α_1.

In all other cases, the intersection is empty (line 19). This includes the case that neither α_1 is a prefix of α_2 nor α_2 is a prefix of α_1 nor $\alpha_1=\alpha_2$ and the case that $\alpha_1=\alpha_2$ and exactly one node of n1, n2 is a leaf node and the other is an inner node.

V. Evaluation

To evaluate our implementation of the Wavelet Trie, we want to compare it with an approach that supports all the desired operations and provides a compression of the string sequences.

A natural approach to compare our implementation of the Wavelet Trie with is the approach to compress string sequences with a standard compressor like gzip or bzip2, to

decompress them on demand, and to do the desired operations on the decompressed string sequence.

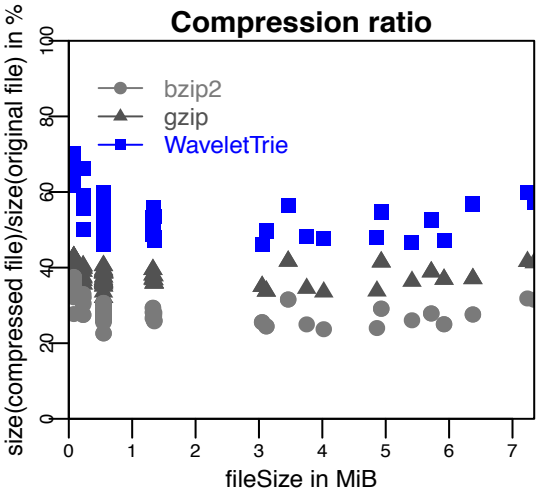

Figure 12. Compression ratio.

We did not compare our implementation with delta-encoding as delta-encoding has the following disadvantage. Delta-encoding cannot support any of the range queries, i.e., our prefix search (III.A), Between, LessThan, and GreaterThan (III.B), because equal strings are encoded different, depending on the previous string. Even intersection (III.F) is not supported. Therefore, delta-encoding does not meet our requirements.

The dictionary-based approach, assigning a segregated Huffman code to each entry results in a bit sequence that supports alphabetical comparisons. Run-length encoding compressing longer bit sequences also supports alphabetical comparisons. Both approaches are orthogonal and compatible to our approach, i.e., can be combined with it. As therefore, a performance comparison with dictionary-based approaches or with RLE is not useful, we have compared our approach with the powerful and widely use compressors gzip und bzip2.

We ran our tests on Mac OS X 10.5.5, 2.9 GHz Intel Core i7 with 8 GB 1600 MHz DDR3 running Java 1.8.0_45.

To evaluate rather text-centric operations, we used 114 texts of the project Gutenberg [15] with file sizes from 78 kB up to 7.3 MB to build a heterogeneous corpus. In order to simulate database operations of a column-oriented database, we used author information extracted from DBLP [16]. Out of these information, we generated lists consisting of 2500 up to 100000 authors.

In all time measurements, we performed 10 redundant runs and computed the average CPU time for all these runs.

A. Compression and Decompression

When evaluating the pure compression and decompression of Wavelet Trie, gzip and bzip2, we get the result that bzip compresses strongest while Wavelet Trie compresses worst (c.f. Figure 12), and that gzip compresses and decompresses fastest while Wavelet Trie compresses and decompresses slowest (c.f. Figure 13).

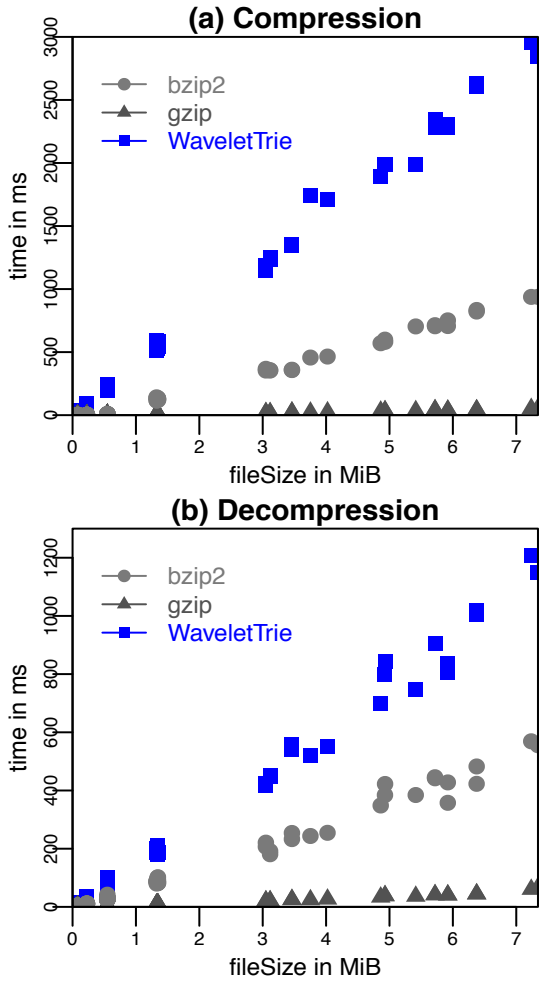

Figure 13. Compression and decompression time.

The main difference between the Wavelet Trie and the generic compressors is that the Wavelet Trie supports many operations on the compressed data, while gzip and bzip2 require to at least decompress the compressed data first, and for some operations to recompress the modified data afterwards. This means, there are a lot of applications that do not require the Wavelet Trie to decompress, as the concerning operations can be evaluated on the compressed data directly. We show the benefit of using the Wavelet Trie in the following subsections, in which we evaluate the performance of the different operations.

B. Search and searchPrefix

Figure 14 shows the search times for (a) a single word and (b) all words starting with a given prefix directly in the Wavelet Trie compared to the time needed for the pure decompression of bzip2 and gzip.

We searched within our Gutenberg corpus for all positions of the word ‘file’, which is contained in each file, and for all positions of words starting with the prefix ‘e’. Although the times for bzip2 and for gzip comprise only the

pure decompression, i.e., no search operation is performed on the decompressed bzip2 or the decompressed gzip file, the search directly on the Wavelet Trie is faster than the pure decompression times of bzip2 and of gzip.

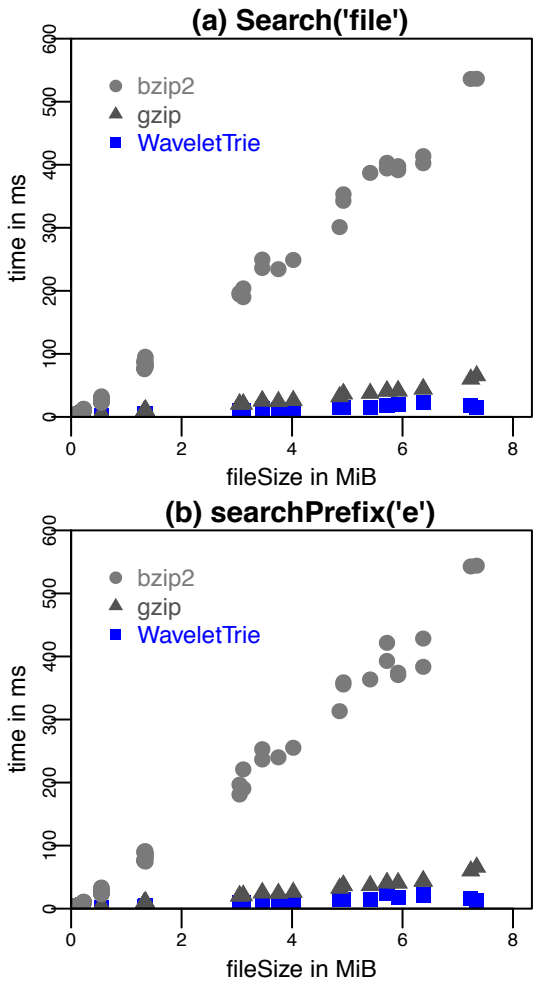

Figure 14. Search times for words in the Wavelet Trie compared to pure decompression time of bzip2 and of gzip.

C. Range queries

Figure 15 shows the results of comparing the search times (a) for words greater than 'e' but less than 'f' and (b) for words greater than 'identification' and less than 'identifier' directly on the Wavelet Trie with the pure decompression time of bzip 2 and gzip. These operations were again evaluated on the Gutenberg corpus. Although the times for bzip2 and for gzip comprise only the pure decompression, i.e., no search operation is performed, the search directly on the Wavelet Trie is faster than the pure decompression times of bzip2 and of gzip. The more specific the search query is, and thus the smaller the search result, the better is the performance benefit of the Wavelet Trie compared to bzip2 and gzip.

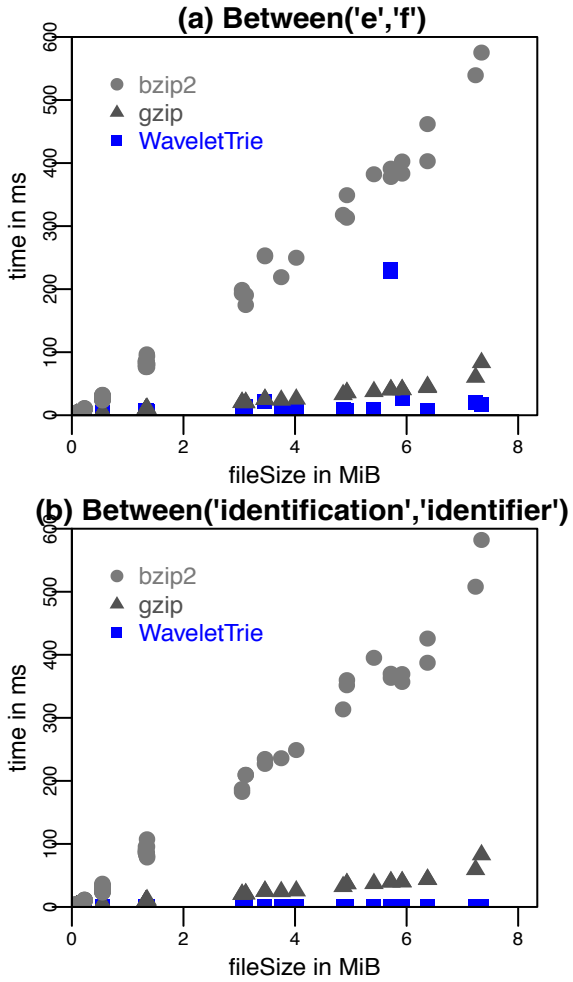

Figure 15. Range queries on the Wavelet Trie compared to pure decompression time of bzip2 and gzip.

D. Insert and Delete

As a first operation, we compared the insert and the delete operation directly on the Wavelet Trie with the pure decompression time of bzip2 and of gzip. We performed these operations on the documents of our Gutenberg corpus. Figure 16 shows the results. The insertion of the word 'database', which does not occur in any of the documents, as 50th word is faster than the pure decompression of bzip2 and as fast as the pure decompression of gzip. The same holds for the deletion of the 50th word.

Note that the decompression times for bzip2 and gzip neither contain the time needed to insert (or to delete respectively) a string nor the time needed to recompress the modified results.

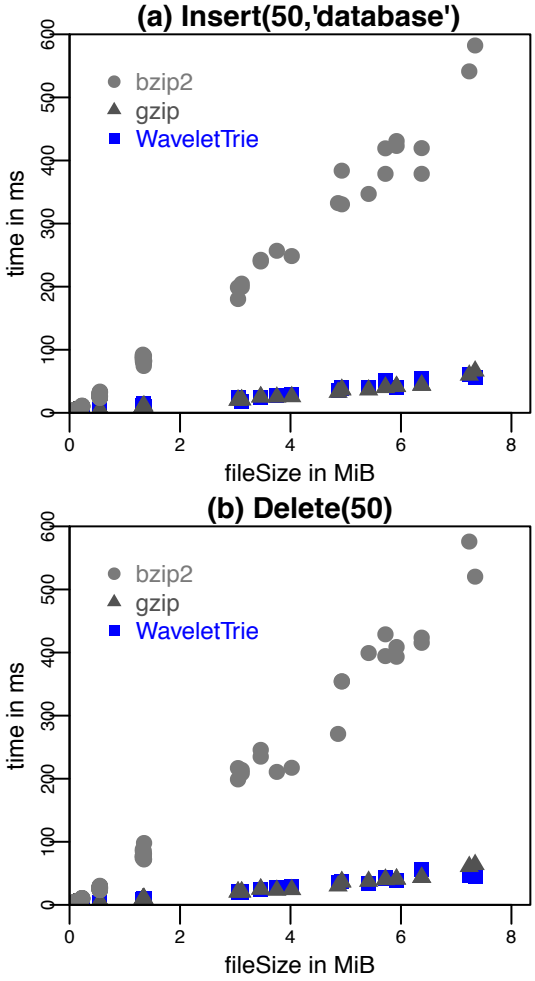

Figure 16. (a) Insertion and (b) Deletion in the Wavelet Trie compared to bzip2 and gzip decompression time.

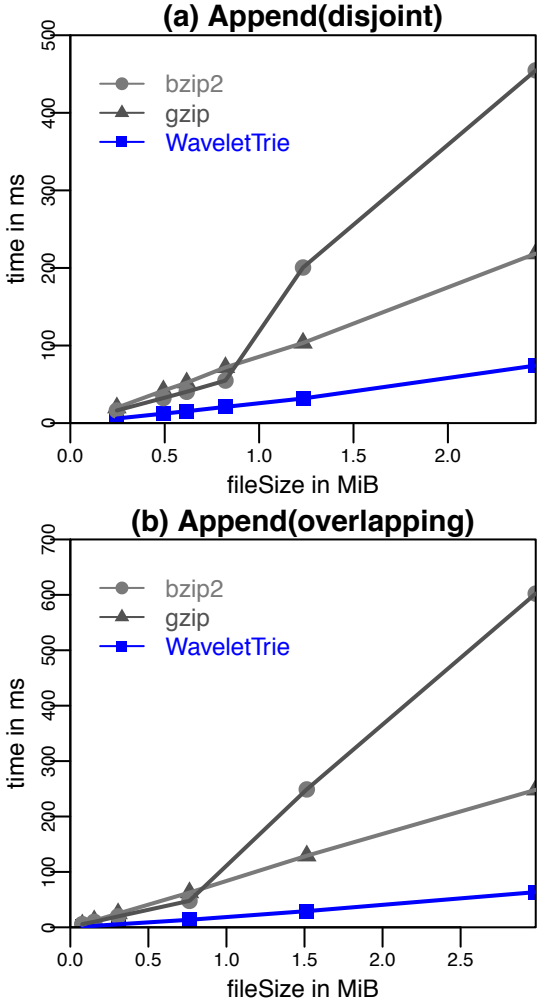

Figure 17. Comparison of the time to append two lists for Wavelet Trie, bzip2 and gzip.

E. Merge/Union

Finally, we evaluated the time to append one list to another list (c.f. Figure 17) and the time to insert a list at position 50 into a second one (c.f. Figure 18).

We performed both tests for disjoint lists as well as for lists that overlap in 50% of the entries. Again, we compared the time with the sequence of decompression and recompressing the concatenated list by using either bzip2 or gzip, i.e., we did not perform any merge or append operation for bzip2 or gzip. In both cases and for both operations, this operation on the Wavelet Trie is faster than the simulation of this operation for bzip2 and for gzip. The benefit of the Wavelet Trie in comparison to bzip2 and gzip is bigger for append operations than for the merge operation that inserts one list at a given position into the second one.

F. Intersection

The following tests were performed on our dblp author corpus. Figure 19 shows the results of comparing the intersection operation on two author lists with the sequence of decompression and recompressing the result list of the intersection using bzip2 and gzip. We computed the result list of the intersection prior to the test runs, i.e., the time needed to compute the intersection was not measured. We used two different sets of lists: the first is duplicate-free, whereas, in the second set, 50% of the list entries of the second list occur also in the first list. If the lists are completely disjoint, the intersection computed directly on the Wavelet Trie is faster than the sequence of decompression and recompression for bzip2 and as fast as this sequence of operations for gzip. If there is a large overlapping of the lists, gzip is faster than the Wavelet Trie, which still is faster than bzip2. Note that we did not perform any intersection operation for gzip and bzip2 compressed data.

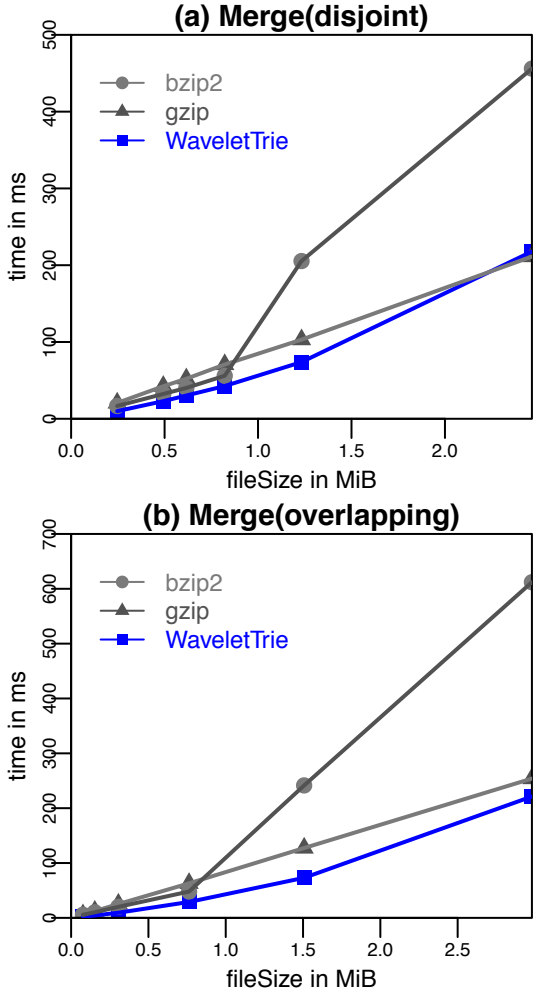

Figure 18. Comparison of the time to merge a list into another one for Wavelet Trie, bzip2 and gzip.

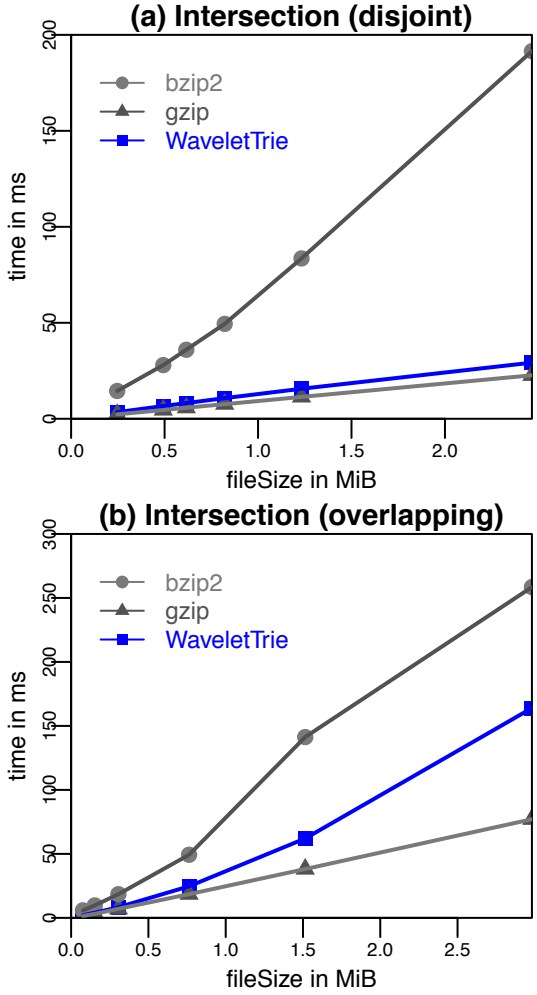

Figure 19. Computing the intersection directly on the Wavelet Trie compared to decompression, list concatenation and recompression time of bzip2 and gzip.

VI. Conclusion

In this paper, we presented and evaluated an extension of the Wavelet Trie [12][13] that allows to represent compressed indexed sequences of strings. As our evaluations have shown, operations like insertion, deletion, search queries, range queries, intersection and union can be performed on the compressed data as fast as or even faster than the simulation of these operations with the help of generic compressors like bzip2 or gzip. We therefore believe that the Wavelet Trie is a good approach to be used, e.g., in column-oriented main-memory databases to enhance the storage or memory capacity at the same time as the search performance.

References

[1] S. Böttcher, R. Hartel, and J. Manuel, "A Column-Oriented Text Database API Implemented on Top of Wavelet Tries," in DBKDA 2017, The Ninth International Conference on Advances in Databases, Knowledge, and Data Applications, 2017, pp. 54–60.

[2] M. Stonebraker et al., "C-Store: A Column-oriented DBMS," in Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005, 2005, pp. 553–564.

[3] A. Lamb et al., "The Vertica Analytic Database: C-Store 7 Years Later," Proc. VLDB Endowment, vol. 5, no. 12, pp. 1790–1801, 2012.

[4] F. Färber et al., "SAP HANA Database - Data Management for Modern Business Applications," ACM Sigmod Rec., vol. 40, no. 4, pp. 45–51, 2012.

[5] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in SODA '03 Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms, 2003, vol. 39, no. 1, pp. 841–850.

[6] D. A. Huffman, "A Method for the Construction of

Minimum-Redundancy Codes," in Proceedings of the IRE, 1952, vol. 40, no. 9, pp. 1098–1101.

[7] T. C. Hu and A. C. Tucker, "Optimal Computer Search Trees and Variable-Length Alphabetical Codes," SIAM J. Appl. Math., vol. 21, no. 4, pp. 514–532, 1971.

[8] S. T. Klein and D. Shapira, "Random Access to Fibonacci Codes," Stringology, 2014, pp. 96–109, 2014.

[9] M. Külekci, "Enhanced variable-length codes: Improved compression with efficient random access," in Proc. Data Compression Conference DCC–2014, 2014, pp. 362–371.

[10] N. R. Brisaboa, A. Fariña, S. Ladra, and G. Navarro, "Reorganizing Compressed Text," in Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, 2008, pp. 139–146.

[11] J. Herzberg, S. T. Klein, and D. Shapira, "Enhanced Direct Access to Huffman Encoded Files," in Data Compression Conference, 2015, p. 447.

[12] R. Grossi and G. Ottaviano, "The Wavelet Trie: Maintaining an Indexed Sequence of Strings in Compressed Space," *CoRR*, 2012. [Online]. Available: http://arxiv.org/abs/1204.3581. [Accessed: Nov, 2017].

[13] R. Grossi and G. Ottaviano, "The Wavelet Trie: Maintaining an Indexed Sequence of Strings in Compressed Space," in Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012, 2012, pp. 203–214.

[14] D. R. Morrison, "PATRICIA---Practical Algorithm To Retrieve Information Coded in Alphanumeric," J. ACM, vol. 15, no. 4, pp. 514–534, 1968.

[15] "Project Gutenberg," 2015. [Online]. Available: http://www.gutenberg.org/. [Accessed: Nov, 2017].

[16] "DBLP: computer science Bibliography." [Online]. Available: http://dblp.uni-trier.de. [Accessed: Nov, 2017].