A Column-Oriented Text Database API Implemented on Top of Wavelet Tries

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

Abstract—**Whenever column-oriented main-memory databases require both, a space efficient storage of strings and an efficient evaluation of operations on these strings, a compressed indexed sequence of strings might be a good choice to fulfill these requirements. A data structure that compresses the string sequence and at the same time supports efficient evaluation of basic read and write operations is the Wavelet Trie. In this paper, we extend the Wavelet Trie by different set-oriented read operations relevant for column-oriented databases like union, intersection and range-queries, and describe how they can be implemented on top of the Wavelet Trie. Furthermore, in our evaluations, we show that performing typical operations on string sequences like searching for exact matches or prefixes, range queries, insert, or delete operations, and operations on two string sequences like merge or intersection, can be performed faster directly on the Wavelet Trie than simulating these operations on bzip2- or gzip-compressed data.**

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

I. INTRODUCTION

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 [1]. 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.

Column stores like, for instance, C-STORE [1], Vertica [2] or SAP HANA [3] 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 is a self-index data structure and can be regarded as an enhancement of variable length encodings (e.g. Huffman [4], Hu-Tucker [5]) that 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 [6] or Elias and Rice variable length encoded data [7]. In [8] 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 [9] 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 [10][11], 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 [12] for string sequences S and the Patricia Trie [13]. 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 this column and that is used for example when finding values of the same database tuples contained in other columns, or *search(s)/searchPrefix(s)* that searches for all positions within the current column that contains the value s (or that have 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.

[10] and [11] introduce the concept of the Wavelet Trie and discuss the complexity of the following operations:

- Access(pos) returns the pos-th string of the sequence

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-558-6

- Rank(s, pos)/RankPrefix(s, pos) return the number of occurrences of string s (or strings starting with prefix s) up to position pos
- Select(s, i)/SelectPrefix(s, i) returns the position of the i-th string s (or string starting with prefix s) of the sequence
- Insert(s, pos) inserts the string s before position pos
- Append(s) appends s to the end of the sequence
- Delete(pos) deletes the string at position pos

which is sufficient to support the most elementary database operations in column stores.

However, in order to support more enhanced data analysis, efficient query processing should go beyond these elementary operations. Here, the main remaining challenge is to support efficient complex read operations like range queries, union, and intersection 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 [10][11] published in 2012 by Grossi and Gupta by concepts and efficient implementations of enhanced database operations (intersection, union, and range queries).

Our second contribution is an evaluation, comparing the performance of the Wavelet Trie with bzip2 and gzip. 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.

In Section 2, we introduce the basic concepts used in the following sections. In Section 3, we explain different operations on the Wavelet Trie and discuss how to implement them. In Section 4, 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 compression.

II. BASIC CONCEPTS

Similarly to [10][11], we define the Wavelet Trie as follows:

Definition (Wavelet Trie): Let S be a non-empty, prefix free sequence of binary strings, $S=(s_0,..., s_n)$, $s_i \in \{0,1\}^*$, whose underlying string set $S_{set}=\{s_0,\ldots,s_n\}$ is prefix-free. The Wavelet Trie of S, denoted WT(S), is built recursively as follows:

(i) If the sequence consists of a single element only, i.e., $s_0 = \ldots = s_n$, the Wavelet Trie is a node labeled with $\alpha = s_0 = \ldots = s_n$.

(ii) Otherwise, let α be the longest common prefix of S. For any $0 \le i < n$, we can write $s_i = \alpha b_i \gamma_i$, where b_i is a single bit. For $b \in \{0,1\}$, we can then define two sequences $S_{\alpha 0} = (\gamma_i \mid b_i=0)$ and $S_{\alpha 1} = (\gamma_i \mid b_i=1)$, depending on whether the string s_i begins with $\alpha 0$ or $\alpha 1$, and the bitvector $\beta = (b_i)$. The bitvector β discriminates whether the suffix γ_i is in $S_{\alpha 0}$ or $S_{\alpha 1}$.Then, the Wavelet Trie of S is the tree whose root is labeled with α and β, and whose children (respectively labeled with 0 and 1) are the Wavelet Tries of the sequences $S_{\alpha 0}$ and $S_{\alpha 1}$.

Remarks:

The requirement that S has to be a prefix free sequence, i.e., no string s_i is allowed to be a prefix of a string s_j, is not a critical restriction, as a prefix free set of strings can be easily constructed for any set S, by adding a terminal symbol not occurring in S to each string $s \in S$.

If the sequence consists of a single element only, we know by the number of corresponding bits within the β of the parent node the size n of the sequence. If the node does not have a parent node, we cannot derive the size of the sequence. In that case, we could either store the size externally or add a binary string s_{new}, $s_{new} \neq s_i$, $i \in \{0,\ldots,n\}$ to the end of the sequence S.

In order to apply the Wavelet Trie to a sequence of words $W=(w_0,\ldots,w_n)$, we compute the sequence of binary strings $SW=(ht(w_0),\ldots,ht(w_n))$, where ht(w) applies the Hu-Tucker algorithm [5] to the word w, yielding a lexicographic Huffman code for w, i.e., $ht(w_i) <_{lexi} ht(w_j) \Leftrightarrow w_i <_{lexi} w_j$, where $<_{lexi}$ denotes "less than in lexicographical order".

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

III. DATABASE OPERATIONS ON TOP OF THE WAVELET TRIE

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

A. Search/Query Operations

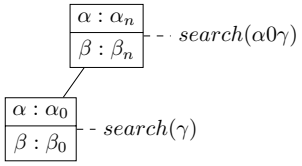

Figure 1. Search operation, if α is a prefix of search string s.

This operation searches for the positions of all binary strings equal to or starting with the given binary string s.

As an auxiliary method, we use the generalization of the select operation to a list of positions: B.select(b, $(p_1,\ldots,p_n)$)=(B.select(b,p_1),…,B.select(b,p_n)), where B.select(b, p_n) denotes the position of the p_n^{th} bit b in a binary string B.

In order to search for all positions of the string s in the Wavelet Trie's root node, we go down the tree representing the Wavelet Trie recursively, until we have found all bits of the binary string s. If we require an exact match, we have found all bits, and we have reached the end of the α of a leaf node, we "translate" the positions of the leaf node into the corresponding positions of the root node with the help of the select operation. That is, if $r=(p_1,\ldots,p_n)$ is the result computed for the n's b-child, then $n.\beta.select(b, (p_1,\ldots,p_n))$ is the result for node n. Similarly, if we search for all positions of binary strings starting with the given prefix s, and 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.

In more detail, the search works as follows: Let the current node n with label α (and β, if n is no leaf node) have a parent node pa, such that n is the b-child of pa and β of pa contains k b-bits and assume that we search for binary strings starting with a prefix s.

If s=α, or s is a prefix of α, we return the list of positions $(1,\ldots, k)$.

Furthermore, if s≠α, α is a prefix of s and n is no leaf node, s is of the form s=αbγ (with a potentially empty γ; c.f. Figure 1). In this case, we perform the search operation for binary string γ on n's b-child, resulting in a list of positions $(p_1,\ldots, p_n)$. In this case, we return the list of positions $n.\beta.select(b, (p_1,\ldots,p_n))$.

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.

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

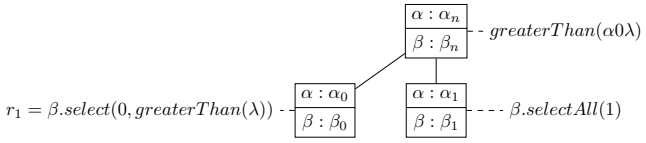

Figure 2. greaterThan if α is a prefix of search string s.

The operation between(s_1,s_2) returns a set of positions of strings r, such that $s_1 \le r \le s_2$. Similarly, the operation lessThan(s) returns a set of positions of strings r, such that r≤s, and the operation greaterThan(s) returns a set of positions of strings r, such that r≥s. In this section, we explain how to implement the operation greaterThan(s). To adapt this operation in order to implement between(s_1,s_2) or lessThan(s) is quite straightforward.

Again, we use as auxiliary operations the generalization, $B.select(b, (p_1,\ldots,p_n)) := (B.select(b, p_1),\ldots,B.select(b, p_n))$, of the select operation to a list of positions. Furthermore, we use the operation $B.selectAll(b) := B.select(b,(1,\ldots,rank(b,|B|))$ which returns the positions of all bits b within the sequence B.

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

If α=s or s is a prefix of α, i.e., α=sbδ (with a potentially empty δ), we know that all strings represented by the Wavelet Trie rooted in n are greater than or equal to s, i.e., we return the set of positions $\{1, \ldots, |\beta|\}$.

In the other case, if α is a prefix of s, i.e., s= αbλ, we have to consider the value of b. If b=1, all strings represented by the Wavelet Trie rooted in n's 0-child are less than s. Therefore, we have to apply the operation r'=greaterThan(λ) to n's 1-child n_1, and return r=β.select(1, r'). If b=0, the result set r consists of two sub-sets r_1 and r_2 with $r=r_1 \cup r_2$ which are computed as follows. As all strings represented by the Wavelet Trie rooted in n's 1-child are greater than s, r_1= β.selectAll(1). Afterwards, we have to apply the operation r'=greaterThan(λ) to n's 0-child, to get r_2= β.select(0, r') (c.f. Figure). Finally, we return $r=r_1 \cup r_2$.

Note that we can similarly create a Wavelet Trie that consists of strings greater than or equal to s, if we do not only return the list of results, but delete all strings not belonging to a result positions.

C. Intersection

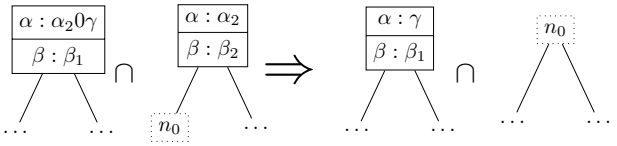

Figure 3. Intersection if α_1 is a prefix of α_2.

This operation 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 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, return n_1 as resulting Wavelet Trie.

If $\alpha_1=\alpha_2$ and n_1 and n_2 are inner nodes, compute the result node r_0 of the intersection of the 0-child of n_1 and of the 0-child of n_2 and the result node r_1 of the intersection of the 1-child of n_1 and of the 1-child of n_2. Then 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.

Let now either α_1 be a prefix of α_2 or vice versa. 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). In this case, we change α_1 into $\alpha_1=\gamma$ and intersect this new node with the b-child of n_2. If after the intersection, the β of the result node contains only 1 bits or only 0 bits, we collapse it with its single child node b-child n_b and thereby delete its (1-b)-child.

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 shows the state before and after collapsing the node for b=0.

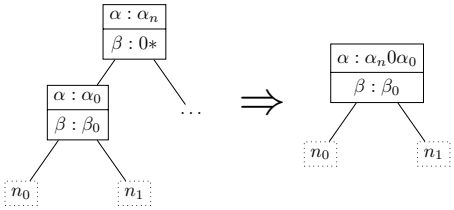

Figure 4. Before and after collapsing the node.

In all other cases, the intersection is empty. 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.

D. Merge/Append + Union

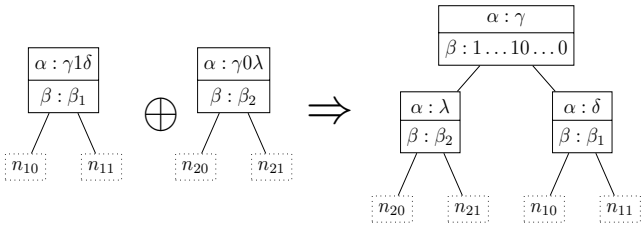

Figure 5. Appending Tries t_1 and t_2 if α_1 and α_2 share a common prefix.

This operation unites/merges two Wavelet Tries t_1 and t_2, i.e., it inserts the string sequence s_2 represented by Wavelet Trie t_2 at a given position pos into the string sequence s_1 represented by Wavelet Trie t_1. Note that this operation is only defined if the set $s_1 \cup s_2$ is prefix free.

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, nothing has to be done, and the operation is finished. If $\alpha_1=\alpha_2$ and both nodes are inner nodes, we insert β_2 at position pos into β_1, merge the 0-child of n_2 at position β_1.rank(0,pos) into the 0-child of n_1 and merge the 1-child of n_2 at position β_1.rank(1,pos) into the 1-child of n_1. Note that the cases that n_1 is a leaf node, but n_2 is not a leaf node, and vice versa, cannot occur, as $s_1 \cup s_2$ is prefix free.

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

If α_2 is a prefix of α_1, i.e., $\alpha_1=\alpha_2 b \lambda$, we create a new node n with labels $\alpha=\alpha_2$ and β consisting of $|\beta_1|$ bits b in which we insert β_2 at position pos. 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. The (1-b)-child of the node n is the (1-b)-child of n_2.

Otherwise, α_1 and α_2 share a common prefix. 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). Then, we create a new node n with labels $\alpha=\gamma$ and β consisting of $|\beta_1|$ bits 1 in which we insert $|\beta_2|$ bits 0 at position pos. 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, 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.

E. Insert/Append

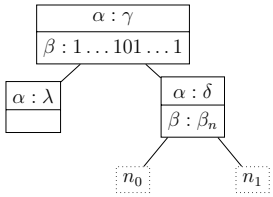

Figure 6. Result of insert operation if s and α_n have a common prefix.

This operation inserts a binary string s into the Wavelet Trie at position pos (or appends it to the end, if pos refers to a position after the number i of entries in the Wavelet Trie.

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 is a prefix of s, i.e., $s=\alpha_n b \delta$, where b is a bit, we insert bit b at position pos into β_n, and insert the binary string δ into the b-child of n at position rank(b,pos).

If n is a leaf node of the Wavelet Trie, and $\alpha_n=s$, we are completed and do not need to do anything else.

Let γ be the common prefix of α_n and s and let us assume w.l.o.g. that $\alpha_n=\gamma 1 \delta$ and $s=\gamma 0 \lambda$. 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 $\alpha=\gamma$, and β consists of $|\beta_n|$ 1-bits and one 0-bit at position pos. The new 0-child of n is a node n' with $\alpha=\lambda$. The new 1-child of n is a node n'' with $\alpha=\delta$ and $\beta=\beta_n$. n'' gets n_0 as 0-child and n_1 as 1-child. Figure shows this case after having inserted s into α_n.

F. Delete

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

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. 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 n's b-child.

If β contains either only 0-bits or only 1-bits (i.e., $\beta=0*$ or $\beta=1*$, in general $\beta=b*$), we have to collapse the current node with its b-child n_b and thereby delete its (1-b)-child as described in Section 2.

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-558-6

IV. EVALUATION

We compared our implementation of the Wavelet Trie with the common compressors gzip and bzip2. 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 (II.A), Between, LessThan, and GreaterThan (II.B), because equal strings are encoded different, depending on the previous string. Even intersection (II.C) 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 [14] 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 [15]. Out of these informations, 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

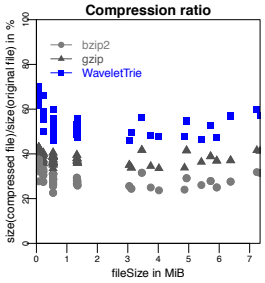

Figure 7. Compression ratio.

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), and that gzip compresses and decompresses fastest while Wavelet Trie compresses and decompresses slowest (c.f. Figure 8).

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

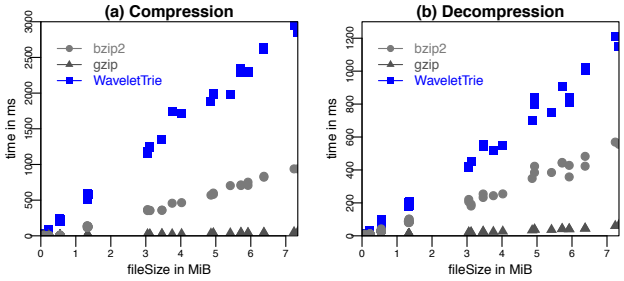

Figure 8. Compression and decompression time.

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. 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 9 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. Please consider that the compression or 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.

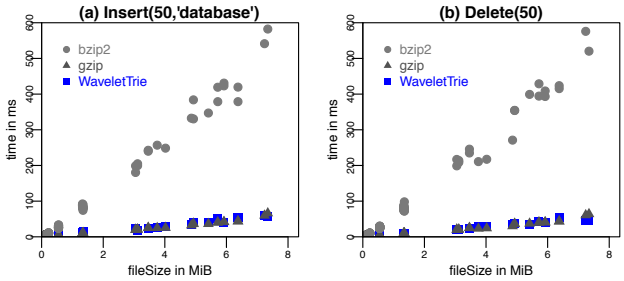

Figure 9. (a) Insertion and (b) Deletion in the Wavelet Trie compared to bzip2 and gzip decompression time.

C. Search and searchPrefix

Figure 10 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 gzip comprise 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

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-558-6

Trie is faster than the pure decompression time of bzip2 and gzip.

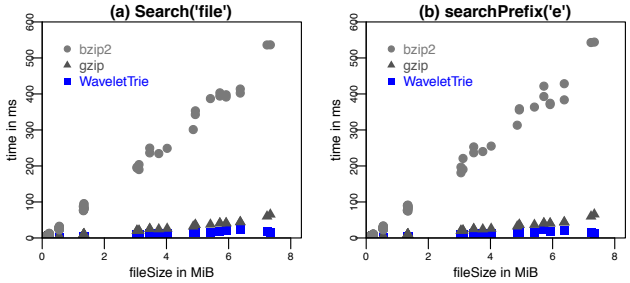

Figure 10. Search times for words in the Wavelet Trie compared to pure decompression time of bzip2 and of gzip.

D. Range queries

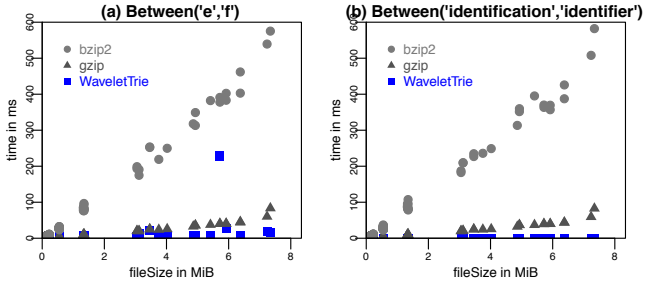

Figure 11 Range queries on the Wavelet Trie compared to pure decompression time of bzip2 and gzip.

Figure 11 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 gzip comprise the pure decompression, i.e., no search operation is performed, the search directly on the Wavelet Trie is faster than the pure decompression time of bzip2 and 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.

E. Intersection

The following tests were performed on our dblp author corpus. Figure 12 shows the results of comparing the intersection operation on two author lists with the sequence of decompression, concatenating the two lists (as we did not want to measure a maybe inefficient string intersection method), 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 simulated operation for bzip2 and as fast as this operation for gzip. If there is a large overlapping of the lists, gzip is faster than the Wavelet Trie, which still is faster than bzip2.

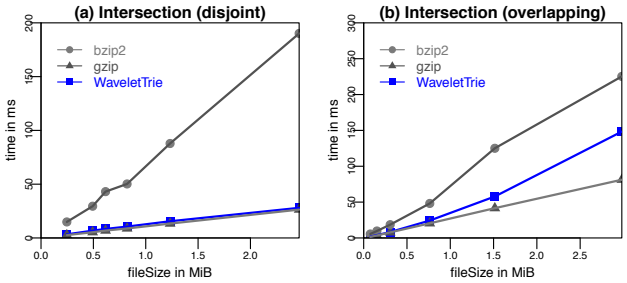

Figure 12. Computing the intersection directly on the Wavelet Trie compared to decompression, list concatenation and recompression time of bzip2 and gzip.

F. Merge/Union

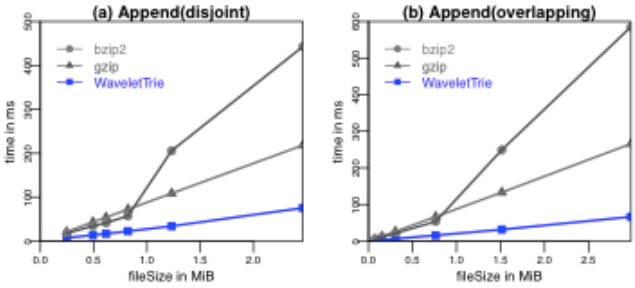

Figure 13. Comparison of the time to append two lists for Wavelet Trie, bzip2 and gzip.

Finally, we evaluated the time to append one list to another list (c.f. Figure 13) and the time to insert a list at position 50 into a second one (c.f. Figure 14).

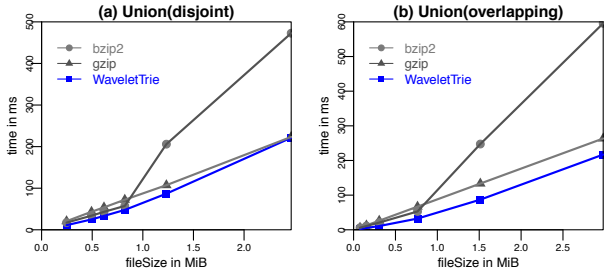

Figure 14. Comparison of the time to merge a list into another one for Wavelet Trie, bzip2 and gzip.

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, concatenating the two lists, and recompressing the concatenated list by using either 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.

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-558-6

V. CONCLUSION

In this paper, we presented and evaluated an extension of the Wavelet Trie [10][11] 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] 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.

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

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

[4] 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.

[5] 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.

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

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

[8] 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.

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

[10] 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: Mar, 2017].

[11] 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.

[12] 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.

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

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

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

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-558-6