The VLDB Journal (2020) 29:1263–1285
https://doi.org/10.1007/s00778-020-00620-x

SPECIAL ISSUE PAPER

Faster & strong: string dictionary compression using sampling and fast vectorized decompression

Robert Lasch[1,4] · Ismail Oukid[1] · Roman Dementiev[2] · Norman May[1] · Suleyman S. Demirsoy[3] · Kai-Uwe Sattler[4]

Received: 14 January 2020 / Revised: 12 June 2020 / Accepted: 16 June 2020 / Published online: 20 July 2020

Abstract
String dictionaries constitute a large portion of the memory footprint of database applications. While strong string dictionary compression algorithms exist, these come with impractical access and compression times. Therefore, lightweight algorithms such as front coding (PFC) are favored in practice. This paper endeavors to make strong string dictionary compression practical. We focus on Re-Pair Front Coding (RPFC), a grammar-based compression algorithm, since it consistently offers better compression ratios than other algorithms in the literature. To accelerate compression times, we propose block-based RPFC (BRPFC) which consists in independently compressing small blocks of the dictionary. For further accelerated compression times especially on large string dictionaries, we also propose an alternative version of BRPFC that uses sampling to speed up compression. Moreover, to accelerate access times, we devise a vectorized access method, using Intel® Advanced Vector Extensions 512 (Intel® AVX-512). Our experimental evaluation shows that sampled BRPFC offers compression times up to 190 × faster than RPFC, and random string lookups 2.3 × faster than RPFC on average. These results move our modified RPFC into a practical range for use in database systems because the overhead of Re-Pair-based compression for access times can be reduced by 2 ×.

Keywords String dictionary · Compression · Re-pair · Vectorization

1 Introduction

Data compression is crucial for in-memory database management systems (IMDBMS) to reduce both memory consumption and processing time. In particular, lightweight dictionary-based compression schemes are typically used for columns storing string data [1–3]. However, such dictionaries consume a significant amount of memory. For illustration, we conducted an analysis of 9 real-world enterprise resource planning (ERP) systems that represent common use-cases of the SAP HANA IMDBMS [4]. A breakdown of the 9 systems is shown in Fig. 1. System 6 has the highest total footprint of 21.6 TB, with 30% of that being taken up by string dictionaries. String dictionaries on average consume 28% of the total memory footprint of the depicted systems, making them the single largest memory consumer in these systems. We propose an improved compression algorithm to reduce the footprint of string dictionaries by up to 50%. This can lead to a significant reduction in the overall memory footprint of such systems.

In SAP HANA, the compression method used for string dictionaries is *front coding*. Front coding exploits redundancy in strings to compress the dictionary by ordering the strings in the dictionary lexicographically, then representing each string using the length of the common prefix with its predecessor and the remainder that is not common with the string's

✉ Robert Lasch
robert.lasch@sap.com

Ismail Oukid
Ismail.Oukid@sap.com

Roman Dementiev
Roman.Dementiev@intel.com

Norman May
Norman.May@sap.com

Suleyman S. Demirsoy
suleyman.demirsoy@intel.com

Kai-Uwe Sattler
kus@tu-ilmenau.de

1 SAP SE, Walldorf, BW, Germany

2 Intel Deutschland GmbH, Neubiberg, BY, Germany

3 Intel Corporation (UK) Limited, London, UK

4 Technische Universität Ilmenau, Ilmenau, TH, Germany

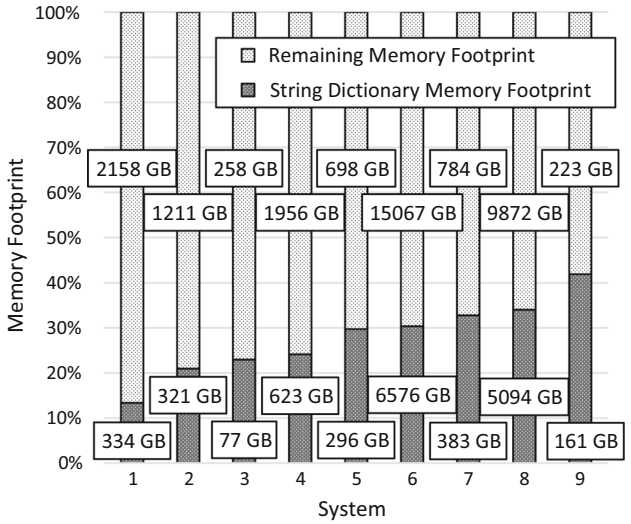

Fig. 1 The memory footprint of 9 ERP systems with the share of the footprint taken by string dictionaries

predecessor. For efficient lookup operations in SAP HANA, the strings are grouped into buckets of 16 strings. This can already achieve significant compression, but as the literature on string dictionary compression shows [5–8], much higher compression rates are possible if the performance of read operations on the dictionary is sacrificed.

Based on this observation, the goal of this work is to find a string dictionary compression method that

(1) significantly increases string dictionary compression in comparison with front coding,
(2) features fast read operations, ideally nearing the speed of front coding,
(3) and has construction times in the same order of magnitude as those of front coding.

The requirements are listed in order of importance. Our first priority is to improve compression (1), but it is very important to maintain reasonable read operation performance (2). Read operations include the extraction of single values, as well as efficient scans of the dictionary. Because string dictionaries are only periodically reconstructed when main and delta are merged as a background task, their construction times (3) are not as important, but should still remain reasonable. Since the related work shows that a tradeoff between requirements (1) and (2) exists, being that increased compression typically mandates slower read operations, we aim to find an existing compression method that already satisfies requirement (1), but has potential to be optimized to also satisfy (2) and (3), if it is not already given.

Comparing different methods for compressing string dictionaries from the literature, performing *Re-Pair* compression [9] on top of front coding consistently offers about two times higher compression rates [7,10] than front coding. Re-Pair is a compression algorithm that compresses its input by creating a grammar consisting of rules that can represent the compressed input in a more compact fashion. Therefore, despite slower access and construction times, we chose this method as the candidate to base this work on and will tackle the aforementioned issues in the process. The simplicity of Re-Pair compression and decompression also makes it a good candidate because it offers several optimization opportunities that have not yet been explored in related work.

Therefore, we aim to improve string dictionary compression, using Re-Pair on top of front coding (RPFC) as a basis, in this work. Our primary goal is to make access operations on string dictionaries compressed with RPFC as fast as possible by better utilizing the computing resources offered by modern CPUs, ideally approaching the access times of string dictionaries compressed with front coding. In addition to this, we aim to achieve faster compression. Of course, while performing the aforementioned optimizations, it is important to sacrifice as little compression quality as possible.

Some of the ideas in this paper were presented in our earlier work [11]. In this paper, we add a detailed analysis of the implementation and the parameters chosen by our compression methods. In particular, we discuss the experiments in more detail that provide evidence that using 16 bits for symbols in the grammar rules is sufficient to capture most potential for compression. We also present new experiments regarding the CPU cache sensitivity of our compression methods. This is an important factor because concurrent workloads contend for precious cache resources. Fortunately, all presented compression methods have a fairly robust runtime behavior when little (shared) last level cache is available. While our earlier work relied on datasets that are widely used in the community to analyze the quality of string dictionary compression, this paper complements this analysis by also sharing results on real-world database columns that store string values. These results are consistent with the results on the datasets used in our prior work.

In this paper, we also extend and refine the block-based RPFC method presented in [11]. The main novel aspect of the new alternative sampled BRPFC is to sample strings from all blocks of the input to create a single shared grammar that is used to compress all blocks of the input data. Because deriving this grammar is the most costly part of the compression step we are able to reduce the compression times compared to BRPFC by 3 × or more. This has no significant impact on the compression ratio or access times.

In Sect. 2, we introduce the basic working principle of the Re-Pair method and survey other compression techniques. The *two major contributions* of this paper, *accelerating construction times* for Re-Pair compressed string dictionaries and *improving their access times*, are presented in Sects. 3 and 4. More precisely, as the first major contribution of the

paper, we accelerate RPFC construction times using a new block-based Re-Pair compression method on top of front coding (BRPFC) (Sect. 3). This also has the advantage of not requiring a single big memory allocation for large dictionaries anymore, but instead making it possible to split the dictionary into manageable blocks of a configurable size. We introduce another improved approach to BRPFC compression that uses sampling to construct the Re-Pair grammar. That approach can further improve construction times at a slight loss of compression quality. As our second main contribution, we utilize SIMD-based optimizations to improve the access times (Sect. 4). Our experiments on well-known text corpora and real-world databases in Sect. 5 demonstrate that these techniques improve compressed dictionary extract times by up to 2.6 × and by 2.3 × on average over the original RPFC decompression method. Our newly proposed sampled BRPFC compression method reaches similar compressed sizes as RPFC at up to 190 × lower construction times and up to 2.6 × faster access times over the original RPFC. The compressed sizes are increased by just 22% on average. Our block-based BRPFC method offers better compression quality than the sampled BRPFC variant with increased compressed sizes over RPFC of just 15% on average. BRPFC requires higher compression times than sampled BRPFC. However, those are still an order of magnitude lower than the compression times of the original RPFC method. In comparison with front coding, sampled BRPFC's *extract* and *locate* operations are only 2.2 × and 1.5 × slower on average, respectively. But sampled BRPFC in turn reduces the average compressed size by 37% over front coding.

2 Background and related work

Vectorized Processing As previous work has identified Re-Pair compression to offer high compression rates but also with high computational costs, we seek to reduce the impact of the computational complexity of this algorithm. Vectorized processing offers a method to accelerate computationally expensive and data-parallel operations. It becomes attractive as more computation has to be performed per byte read from memory [12] as the same operation is applied to multiple operands executing one instruction. On the Intel platform, the SSE and AVX instruction set extensions realize vectorization where the newest generation—AVX-512—offers 512-bit wide registers allowing to process, e.g., up to 16 32-bit integers with one instruction [13]. An interesting pair of instructions introduced in AVX2 is the gather and scatter instructions. They introduce the capability to load or store multiple data elements based on index positions. However, the latency of the load and store operation is determined by the slowest load or store for all operands. Using instruction level parallelism, working on fewer operands, may hide this latency as multiple load or store operations can be executed out of order. A notable extension of AVX-512 to its predecessor AVX2 is the support of a mask for most instructions which allows for a selective execution of the instruction based on the bits set in the mask register. Compilers of higher-level languages such as C++ can automatically emit vector instructions in special cases, such as vectorizing a loop that has no dependencies between iterations. This is called *auto-vectorization* and is relatively unreliable, as it does not apply to most code. This leaves it to the programmer to use vector instructions manually where appropriate, which we are going to do in this paper. To use vector instructions directly in C++, one has to use *intrinsics*, which are built-in C++ functions that are compiled directly to their corresponding vector instructions. A full reference of all AVX-512 intrinsics, including the ones used in this paper, can be found at [14].

String Dictionaries Dictionary compression or domain encoding is a popular method in database systems to reduce their memory footprint [1–3]. The dictionary maps values of the domain of a (set of) column(s) to value IDs. In the records, these values are replaced by the corresponding value IDs which may lead to space savings—especially for strings—and enables efficient processing on dictionary-encoded values [2,15,16]. The string dictionary then offers two access operations: (1) Given a value ID *id*, *extract*(*id*) returns the corresponding string in the dictionary, and (2) Given a string *str*, *locate*(*str*) returns the unique value ID of *str* if *str* is in the dictionary or the value ID of the smallest string greater than *str* otherwise.

Compression of String Dictionaries Müller et al. [8] surveyed several compression methods for string dictionaries to evaluate the tradeoff between compression ratio and access performance: These included front coding, Huffman-Tucker coding, N-Gram compression, and Re-Pair compression to name the most important ones. They also proposed a *compression manager*, which can automate the decision of the compression method for a string column based on the characteristics of the column. In [17], a dictionary-based compression approach is proposed that encodes arbitrary keys in an order-preserving way. This technique is applied to various in-memory search trees. In contrast, our Re-Pair approach aims at compressing columnar string data.

Plain Front Coding (PFC) is a light-weight method to compress sorted string dictionaries. Strings are represented by the length of the common prefix with their predecessor in the dictionary and the remaining suffix. This is illustrated in the second row of Fig. 2. In the figure, each box is the representation of one string in the dictionary. With the front-coded representation, the last two strings are compressed. The first one is not, as it has no predecessor. PFC operates

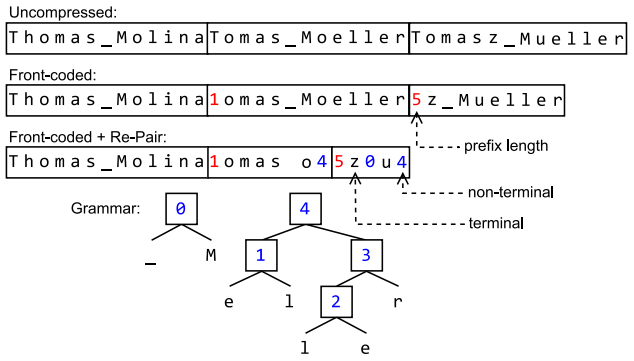

Fig. 2 Example of PFC and RPFC

on buckets of, e.g. 16 strings. The first string of each bucket (the bucket header) is stored uncompressed. This facilitates random accesses to the string dictionary. Without buckets, each access would require decoding all prior strings in the dictionary. With buckets, strings only have to be decoded starting from the beginning of the bucket the desired string is contained in. PFC offers an interesting trade-off between reasonable compression rates and fast access operations and therefore is used in SAP HANA. However, amongst the various methods surveyed by Müller et al., Re-Pair compression, discussed below, on top of PFC, exhibits the best compression ratios.

Re-Pair [9] is a grammar-based compression method. It iteratively obtains a grammar from an input text T and transforms the text into a compressed sequence of symbols. We call any symbols that can be present in the input text T *terminals* and any new symbols that are added during compression *non-terminals*. For our purposes, symbols in the input text are always bytes, and hence terminals are symbols from 0 to 256. This distinction comes from the context of formal grammars, where terminals are symbols that make up the set of strings in the formal language defined by the grammar. Non-terminals cannot be part of those strings. Likewise, with Re-Pair, the compressed sequence may contain non-terminals, but decompressing it, one obtains the input text again, which contains only terminals. The input text can be seen as the only string in the formal language defined by the Re-Pair grammar, when given the compressed sequence of symbols to expand.

In each iteration, Re-Pair first finds the most frequent pair of symbols ab in T. If multiple pairs with the same frequency exist, one candidate is choosen at random. Then, it adds a rule $R \rightarrow ab$ to the grammar, where R is a non-terminal symbol that does not previously appear in T or in the grammar and a and b can be any symbols (terminal or non-terminal). Third, it replaces every occurrence of ab in T by R. This iteration terminates when there are no pairs of symbols left that occur more than once in T. In practice, several auxiliary data structures are used to implement the iterative steps efficiently [9]. Re-Pair on top of PFC (*Re-Pair Front Coding (RPFC)*), introduced by Navarro et al. [7], is the baseline of this work. The third row of Fig. 2 shows how Re-Pair is applied on top of PFC. RPFC applies Re-Pair on top of all front-coded buckets in the dictionary, but the bucket headers are left uncompressed. One grammar is created for the whole dictionary. In Fig. 2, this grammar is visualized as a set of binary trees, where the inner nodes are non-terminals, and the leaf nodes are terminals.

Navarro et al. [7,10] introduced and evaluated techniques besides RPFC for compressing string dictionaries. These included hash-table-based methods and Huffman coding on top of PFC or on the bucket headers of RPFC. Grossi and Ottaviano [6] investigated the use of tries to compress strings by storing identical prefixes only once. The tree relies on pointers to child nodes which requires additional space. In practice, several different methods exist to store tries in a space-efficient and access-friendly manner, e.g. [5,6,18]. Kanda et al. [19] introduced the term "auxiliary string dictionaries" for dictionaries that are used to further compress the internal strings obtained from a front-coded or a decomposed trie dictionary representation, as it is done with Re-Pair for RPFC. *Fast Static Symbol Table* (FSST) [20] is a new lightweight string compression method that was not published yet at the time of writing this paper. It can be used for string compression in general and string dictionary compression in particular. Being a lightweight method, FSST does not aim to provide compression as strong as Re-Pair based methods, but claims faster compression and decompression operations than the methods that will be presented here.

In this paper, we base our work on RPFC because none of the compression methods discussed above consistently dominates RPFC regarding compression ratio and access performance. However, RPFC exhibits substantially higher dictionary access times over PFC (the method currently used in SAP HANA). We show in Sect. 4 how access operations on RPFC can be accelerated on modern CPUs.

3 Block-based re-pair compression

The RPFC compression algorithm [7] has serious shortcomings when targeting large string dictionaries. Primarily, its compression times are very high in comparison with lightweight compression methods such as front coding, growing super-linearly with input size. RPFC compression times quickly become impractical once the input exceeds a few 100 MBs. Secondly, it requires a large amount of memory (multiples of the input text size) for its auxiliary data structures (see Sect. 2). In our use case, dictionary compression is done as a background job during normal IMDBMS operation as a part of merging main and delta storage [4]. Therefore, it is undesirable to require large amounts of memory for this

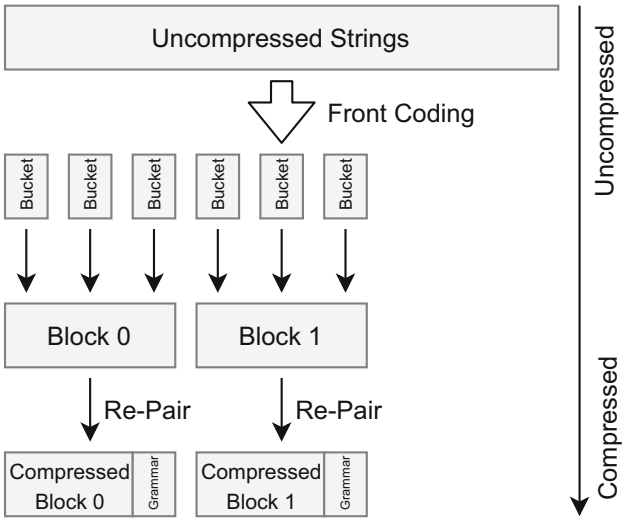

Fig. 3 Example of BRPFC compression

process, as our main motivation is *reducing* the memory footprint. In the worst case, these memory requirements can even make applying RPFC compression impossible, if the memory is simply not available in the system.

In a micro-architecture analysis [21] of the RPFC compression method conducted using Intel® VTune™ Amplifier [22], we found that 70% of the runtime is spent updating and accessing the auxiliary data structures. During these operations, execution stalls roughly 45% of the time due to memory latency. In the update of the text itself (10% of total runtime), 75% of the execution is memory bound. Based on this observation, one can assume that the compression times would significantly improve if the working set of the algorithm fits into CPU caches. Therefore, we propose *Block* Re-Pair Front Coding (BRPFC), which consists in splitting the dictionary into *blocks* containing several front-coded buckets before applying Re-Pair on each block individually. The idea of creating blocks of text to reduce the compression time was also proposed by Gagie et al. [23] concurrently to our initial work. The block size in our work can be chosen such that both the input text and the auxiliary data structures used for Re-Pair compression fit into the CPU caches. We evaluate the effects of compressing at varying block sizes in Sect. 5.

An example of how BRPFC compresses a string dictionary is shown in Fig. 3. A BRPFC-compressed string dictionary with front coding bucket size b is constructed similarly to a RPFC-compressed string dictionary: First, we apply front coding for each b strings in the lexicographically sorted dictionary, yielding $\lceil \frac{N}{b} \rceil$ front-coded buckets of strings, where N is the total number of strings in the dictionary. In contrast to RPFC and to our previous work [11], we explicitly encode the length of the remaining front-coded strings, instead of zero-terminating them. This facilitates storing binary strings in the dictionary, which may contain the symbol 0. A disadvantage of this is a small amount of additional overhead for strings where multiple symbols are required to encode the length. Then, these buckets are concatenated into blocks, omitting the bucket headers (the first string of each bucket). The number of buckets per block can be determined in different ways. The simplest option is to use a fixed number B of buckets per block. This is what is shown in the example in Fig. 3. However, this will result in inconsistent numbers of symbols per block, as the number of symbols per bucket varies depending on the front-coded length of strings in the bucket. It is also possible to use a varying number of buckets per block, such that each block comes as close as possible to a target symbol count s. We evaluate only the latter option in Sect. 5, as it produces much more consistent blocks in terms of the number of symbols present in each block. After the blocks have been formed, each block is then compressed using Re-Pair such that buckets remain self-contained, i.e., no symbol belongs to two adjacent buckets. This is important, as it enables storing integer offsets to each bucket in the compressed text, which facilitate random accesses to the dictionary at decompression time. Note that this is different from the original RPFC algorithm, which enforces the stronger requirement of no symbol belonging to two adjacent *strings*. Thereafter, we write the result as a contiguous sequence of the resulting grammar followed by the buckets reunited with their headers. In this sequence, the grammar is stored using an implicit encoding that stores two symbols per grammar rule. These two symbols are the ones that the rule maps to, while the rule symbol itself is encoded implicitly by the position of the symbols in the sequence. The symbols are stored using a fixed bitwidth of 16 bits per symbol. In contrast to this, the Re-Pair compressed buckets are stored using bit-packing, i.e., using the minimal number of bits per symbol possible given the number of distinct symbols in the block. For example, if the input strings have 256 distinct symbols, and 2000 grammar rules are created by the Re-Pair algorithm, each symbol would be stored using $\lceil \log_2(256 + 2000) \rceil = 12$ bits. We choose not to use bit-packing for the grammars at the expense of some compression quality in favor of decoding performance, as experimentation has shown that compression quality is not improved much by bit-packing the grammars. This will be discussed in more detail in Sect. 4. We store the offsets to the beginning of each bucket in a separate array, similar to RPFC. As BRPFC introduces a new indirection layer, we need to keep offsets to the blocks and their respective grammars as well. This process results in a set of compressed blocks, each storing a multiple of b strings, except for the last block, which can possibly contain a non-multiple of b strings if N is not a multiple of b.

Since Re-Pair is applied on each block separately, its input text length can be adjusted by changing B or s. As noted before, using s and thereby a varying number of buckets per block, yields better control over the input size to the Re-Pair

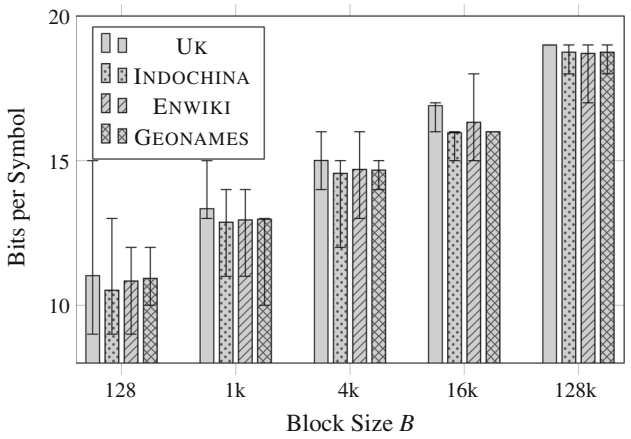

Fig. 4 Bits per symbol in BRPFC compressed dictionaries before fixing bits per symbol to 16

algorithm than using a fixed number of buckets per block. Consequently, picking a suitable block size should allow for Re-Pair's working set to fit into the CPU caches, dramatically reducing compression times. We evaluate compression performance at different block sizes in Sect. 5 to validate our assumption that a low enough block size speeds up compression significantly.

Figure 4 shows how the choice of a certain block size B affects the Re-Pair part of compression. It shows how many bits are required to be used per symbol when compressing four different datasets. Information about the datasets can be found in Sect. 5. Bars show the average number of bits per symbol across all blocks of the dictionary. Whiskers show the minimum and maximum values found in individual blocks of the dictionary. At higher block sizes, Re-Pair replaces more pairs by new symbols than at lower block sizes. Fewer symbols lead to fewer bits needed to represent a single symbol, which can be seen in Fig. 4. As the Re-Pair algorithm does fewer replacements at lower block sizes, compression is also going to be worse compared to original RPFC where the dictionary is compressed as a whole.

Another advantage of BRPFC compression is that it is trivially parallelizable if a fixed number of buckets per block is used, since each block is completely independent of other blocks then. If B is larger than the number of available CPU threads p, then the compression can be sped up by a factor $\Theta(p)$. When using a variable number of buckets per block, it would be necessary to create all blocks first in a sequential manner, to then run Re-Pair compression in parallel for several blocks. We do not explore multi-threading because dictionary compression is typically a background task with compute resources limited to a single CPU thread such that the remaining threads can execute foreground user queries. However, this approach can be a big advantage of BRPFC over RPFC in scenarios where more resources are available.

Table 1 Overlap of rules in BRPFC grammars (block size $s = 8$ M)

	# Total rules n	# Unique rules n_u	Overlap (%)
UK	2,372,492	954,040	59.8
INDOCHINA	1,028,809	570,330	44.6
ENWIKI	2,367,402	527,758	77.7
GEONAMES	456,946	155,801	65.9

Sampled block-based re-pair compression

BRPFC improves compression times over RPFC by running the Re-Pair algorithm on smaller blocks of the dictionary. However, we observe that there is a significant amount of overlap in the grammar rules of BRPFC blocks. Table 1 demonstrates this by reporting the total number of rules n across all grammars of compressed BRPFC dictionaries using the same datasets as Fig. 4, and relating those numbers to the number of *unique* rules n_u across all grammars. These two numbers can be used to determine the percentage of rules that are *not unique* to one grammar, i.e., rules that *overlap* with rules from other grammars, as follows: $\frac{n-n_u}{n}$. This is shown in the last column of Table 1 and demonstrates the significant overlap of grammar rules mentioned above, which lies between 44 and 78%, depending on the dataset. Eliminating this overlap could improve the compression quality of BRPFC. Additionally, if it is not necessary to perform full Re-Pair compression on the whole dictionary, but only on some smaller part of it, compression would be greatly accelerated over BRPFC. This is because the Re-Pair step is by far the most computationally expensive one in the BRPFC compression pipeline.

To eliminate grammar overlap and speed up compression times, we propose a technique for *sampled* BRPFC compressed string dictionary construction. The technique's main steps are shown in Fig. 5. The general idea is to create a single Re-Pair grammar after the front-coding stage. That grammar can then be used to compress all blocks of the dictionary. The grammar is created by sampling a certain number of the front-coded buckets to form a *superblock*. Varying target sizes can be set for the number of symbols in the superblock to improve the resulting grammar at the cost of increasing compression time. Front-coded buckets are chosen for sampling following a base 2 van der Corput sequence [24] that is scaled and discretized to fit the interval of available buckets, and sampling is stopped when the target size has been reached. In our case, the base 2 van der Corput sequence is generated by splitting the interval of all buckets in half recursively. At each recursive step, the split point is selected as an entry of the sequence, and thereby the corresponding bucket is included in the sample. For example, if there were 32 buckets numbered from 0 to 31, the first bucket selected

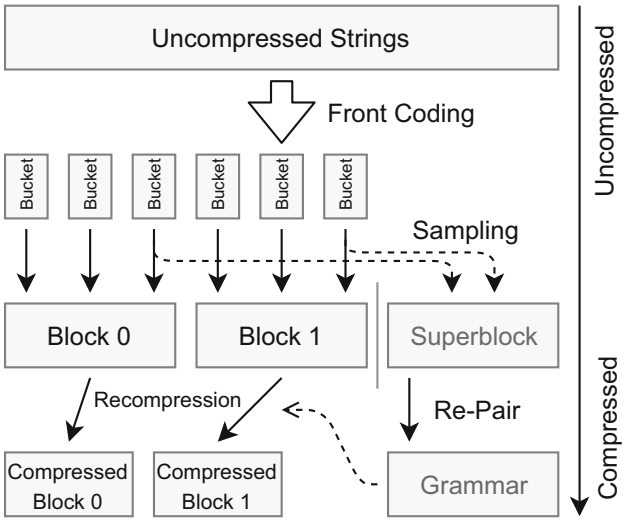

Fig. 5 Overview of sampled BRPFC compression

would be 16, splitting the sequence into two subsequences (from 0 to 16 and from 16 to 31). Then, the centers of each of the subsequences would be selected, i.e., first 8 and then 24, again splitting the subsequences into two subsequences each. This would continue recursively, selecting bucket 4, 12, 20, 28, 2, 6, 10, 14, 18, 22, 26, 30, etc., until the accumulated size of the sampled buckets reaches the superblock target size. Following this sequence results in roughly equal-spaced samples from the set of front-coded buckets, without requiring prior knowledge of how many buckets need to be sampled to reach the target size. This way the sampling rate does not need to be an explicit parameter, but is implicitly determined by the target superblock size. The technique is necessary because the number of sampled buckets cannot be accurately determined in advance using the superblock target size, as buckets vary in size. After sampling, the superblock is compressed using the Re-Pair algorithm, and the resulting grammar is used as the shared grammar.

Next, we apply an algorithm that we call *recompression* to the blocks of the dictionary. Its inputs are the shared grammar and an uncompressed block, and the algorithm outputs a compressed block. This means the algorithm uses the rules from the existing grammar to compress the block. To do this, we first build a trie containing all expanded rules from the shared grammar. The trie is constructed as follows: The expanded version of each grammar rule is constructed first. This can be done by iteratively expanding the rules in the order that they were added to the grammar. Rules that expand to other rules (i.e., non-terminals) can look up their previously expanded child rules. With the rules in expanded form, the actual trie is built. We use a simple node-based representation of the trie in our implementation. Each node can point to as many other nodes as there are distinct symbols in the expanded rules. Expanded rules are added to the trie by starting with the first symbol of the rule at the root node, and traversing the trie nodes based on the symbols in the expanded rule, adding new nodes where necessary. Finally, nodes in the trie that correspond to an expanded rule—where the aforementioned traversal ended after handling the last symbol—are tagged with the non-terminal symbol of that rule. With the grammar trie available, each block is compressed using that representation of the shared grammar. At each position in a block's text, the trie is traversed using symbols from the text. As a result of this, several rule expansions may be found to match the string at the current position in the text. The longest one is selected, and that segment in the text is replaced with the rule's non-terminal. This repeats at the next position after the replaced segment in the text and continues until the whole text has been compressed. If no match in the expanded rules is found, the algorithm continues to the next position in the text. The matching part of the algorithm is similar to the Aho-Corasick [25] string-matching algorithm, the difference being that we only need to find the longest possible expanded string at any position in the text, while Aho-Corasick finds all occurrences of all searched strings in the text. Note that the first part of the recompression algorithm—constructing the grammar trie—only needs to be done once for the whole dictionary, as the shared grammar does not change after it has been initially derived from the superblock.

In comparison with BRPFC, block compression becomes a two-stage process with sampled BRPFC. A shared grammar is constructed first, then blocks are compressed using that grammar. While the Re-Pair algorithm is still used to obtain the shared grammar, it is not required anymore for compressing the blocks. Those are compressed with the recompression algorithm, which is much less computationally expensive than the full Re-Pair algorithm. In sum, this can reduce total compression time significantly. Compression quality can be traded off for compression speed by reducing the target size of the superblock, we evaluate this in Sect. 5.

4 Vectorizing re-pair lookup

The main bottleneck when accessing strings in compressed string dictionaries employing Re-Pair compression is the expansion of non-terminals into the terminals they represent by performing lookups in the Re-Pair grammar. From now on, we assume that this functionality is implemented in the function ExpandSymbol, which expands a single non-terminal to the respective terminals using a Re-Pair grammar. Therefore, we explain in this section how we vectorize ExpandSymbol to improve the overall performance of string access operations in Re-Pair compressed string dictionaries. As Re-Pair expansion is always required for decompression when Re-Pair compression was used, this

applies to RPFC as well as to BRPFC and to sampled BRPFC dictionaries.

Iterative symbol expansion

In the RPFC implementation by Navarro et al. [7], ExpandSymbol is implemented using a recursive algorithm. Since that original implementation of ExpandSymbol is not vectorization-friendly, we first turn the algorithm from a recursive algorithm into an iterative algorithm and also remove as much branching as possible. We show the optimized scalar code in Algorithm 1. N and n represent the total number of non-terminals and terminals, respectively.

Algorithm 1 Simplified iterative Re-Pair symbol expansion

```
1: function ExpandSymbol(symbol, grammar, outputBuffer)
2:     current ← symbol
3:     stack ← buffer of size 32
4:     pos ← 0, stackPos ← 1
5:     do
6:         if current ≥ n then                          ▷ expand non-terminal
7:             left ← GetField(grammar,
                   ⌈log2(n + N)⌉, 2 · (current − n))
8:             right ← GetField(grammar,
                   ⌈log2(n + N)⌉, 2 · (current − n) + 1)
9:             current ← left
10:            stack[stackPos++] ← right
11:        else                                         ▷ expand terminal
12:            outputBuffer[pos++] ← current
13:            current ← stack[−stackPos]
14:        end if
15:    while stackPos > 0
16:    return pos
17: end function
```

ExpandSymbol performs a depth-first traversal of the binary symbol tree, outputting leaves (terminals). We do this iteratively by processing one tree node in each iteration until symbol expansion has finished, using a stack to track nodes that need to be visited at a later point. The current node is stored in `current`, which is initialized to ExpandSymbol's input `symbol`, the root of the symbol tree. Each node encountered can either be a terminal (leaf node) or a non-terminal (inner node). If the node is an inner node (branch 1, lines 7–10), its left and right child nodes are fetched using GetField, which decodes the symbols from the bit-packed grammar array. The right child node is put onto a stack for later use, while the left one becomes `current` for the next iteration. This way, the left path in the tree is always traversed until a leaf is encountered. When this happens (branch 2, lines 12–13), the leaf node (which is a terminal) is output, and a node is popped from the stack to be processed in the next iteration. Once the last leaf node is popped from the stack and output, the tree has been fully traversed, which means that `symbol` has been fully expanded to terminals.

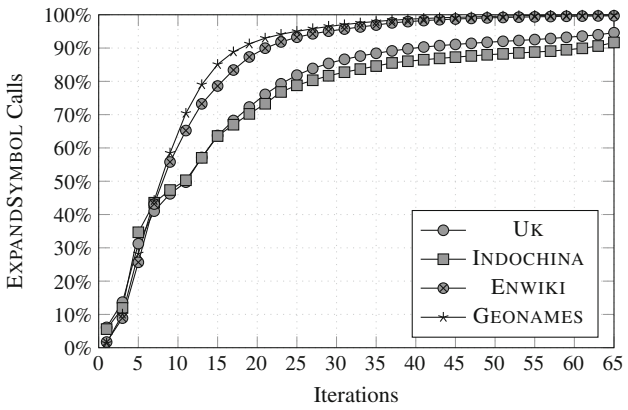

Fig. 6 Upper bound of iterations per scalar ExpandSymbol call before limiting non-terminals to expand to at most 8 terminals

Vectorization prerequisites

To accelerate Re-Pair expansion beyond the aforementioned optimizations, we aim to execute multiple ExpandSymbol operations in parallel using vector instructions from the AVX-512 instruction set extension. As noted earlier, ExpandSymbol consists in doing a depth-first traversal of the grammar tree. Since the latter can be of arbitrary depth, traversals that are executed simultaneously must have a similar number of iterations for the vectorization to be efficient. Otherwise, the vectorization would become inefficient due to the vector elements that require a small number of iterations being unused, while vector elements that require a large number of iterations are still being processed. Therefore, we propose *limiting to 8 the number of terminals a non-terminal can expand to*, which is equivalent to limiting the number of leaves in the symbol trees, thereby allowing us to execute the vectorized code for a fixed number of 15 iterations, which results in a salient performance improvement as discussed in Sect. 5.

The reason for setting the limit to 8 terminals, and in turn requiring 15 iterations per expansion, can be observed in Fig. 6. The figure shows the percentage of ExpandSymbol calls that require at most the number of iterations on the x-axis before the aforementioned limitation is imposed. It can be seen that before imposing the limitation, on average 73% of calls to ExpandSymbol require less than or exactly 15 iterations. As the curves' slopes fall off after 15 iterations, increasing the number of iterations beyond 15 would only increase this percentage slightly, but reduce the performance of the vectorized symbol expansion. Conversely, reducing the number of iterations further drastically cuts down the number of symbol pairs that can be replaced by the Re-Pair algo-

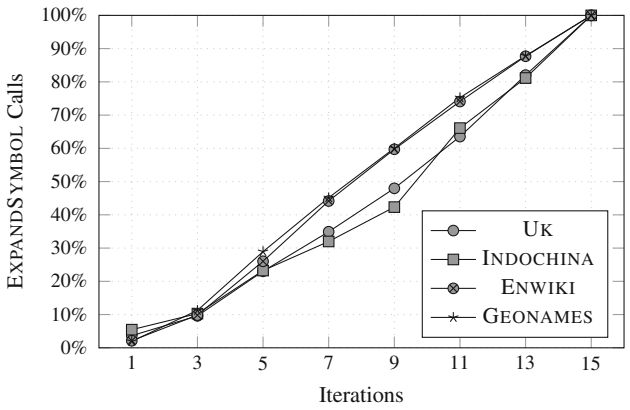

Fig. 7 Upper bound of iterations per scalar ExpandSymbol call after limiting non-terminals to expand to at most 8 terminals

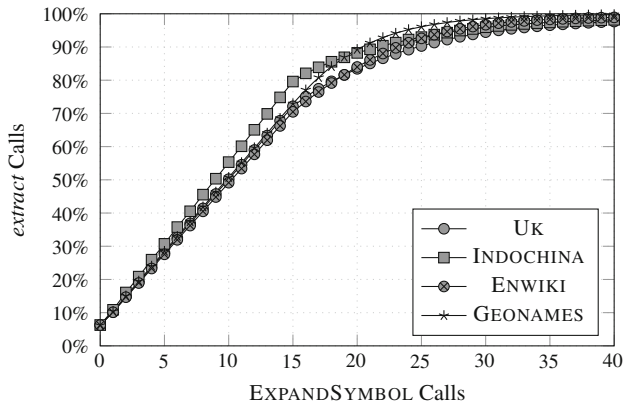

Fig. 8 Upper bound of ExpandSymbol calls necessary per *extract* call before limiting non-terminals to expand to at most 8 terminals

rithm at compression time, thereby reducing compression rates. We therefore consider setting the limit to 8 terminals as the most practical tradeoff between effective compression and fast decompression. The distribution of iterations per ExpandSymbol call *after* imposing the aforementioned limitation can be observed in Fig. 7.

It should be noted that the type of vectorization proposed here can only improve performance if the operation that we are parallelizing is required to be executed multiple times. To validate this, we analyzed how many times ExpandSymbol is called when performing *extract* operations. We show the results in Fig. 8. The figure shows the percentage of *extract* calls that require at most the number of ExpandSymbol calls shown on the x-axis. It can be seen that roughly 75% of *extract* operations requires between 0 and 15 symbol expansions. However, after limiting symbols to expand to at most 8 terminals, the distribution becomes more flat, as shown in Fig. 9. This means that executing multiple symbol expansions simultaneously for a single *extract* operation does make sense, as our analysis shows that a majority of *extract* operations require a substantial amount of symbols to be expanded.

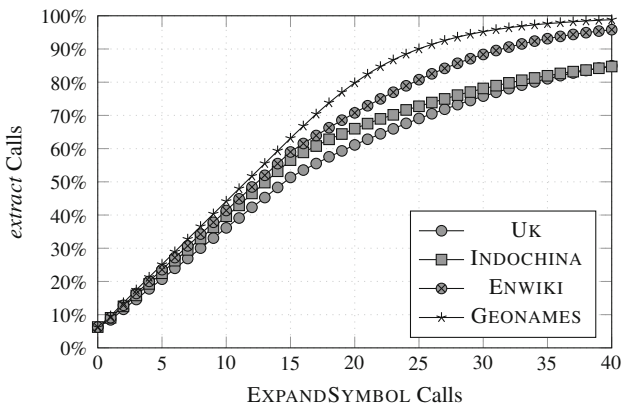

Fig. 9 Upper bound of ExpandSymbol calls necessary per *extract* call after limiting non-terminals to expand to at most 8 terminals

It should also be noted that both before and after introducing the limitation, exactly 6.25% of all *extract* calls do not require any symbol expansions. This is expected and due to the fact that headers of the front coded buckets in the dictionaries are stored uncompressed. As we are using a bucket size of 16, this means that every 16th string is left uncompressed and hence does not require any ExpandSymbol calls.

Additionally, to reduce the computation in each loop iteration, we propose to *fix the number of bits per symbol in the Re-Pair grammar to 16*, thereby avoiding the cost of decoding the bit-packed grammar array. This allows symbols to be loaded directly from the grammar array into CPU registers without bit-offset calculations and expensive bit-shifting. Put differently, the GetField calls in Algorithm 1 can be replaced by simple array access operations if symbols in grammars are 16 bits wide. However, using 16 bits per symbol may have two distinct negative effects:

1. The maximum number of symbols is limited to 2^{16}. This can be less than the number of symbols that would have been generated by the Re-Pair algorithm normally, reducing compression quality if these symbols would have replaced pairs with frequencies high-enough to justify increasing the number of bits to represent them.
2. If $\leq 2^{16-1}$ symbols are used in the generated grammar then less than 16 bits would be needed to represent all symbols. Therefore, using 16 bits to store these symbols wastes space.

This change is therefore a trade-off between compression ratios and decompression performance. Note that this fixed-width encoding could also be used for symbols in the compressed buckets in the compressed dictionary. This was done in our previous work [11]. However, as these symbols need to be decoded less frequently, we decide to store them

```
uint32_t RePair::expand16Symbols(
    uint16_t* symbols, uchar* str) {
    <...> //initialization
    for (size_t i = 0; i < 15; i++)
    {
        <...> //branch 1
        <...> //branch 2
        <...> //predicate evaluation
    }
    <...> //result output
}
```

Listing 1 Overview of EXPAND16SYMBOLS

with variable bitwidths here, which is beneficial for the compression quality.

It should be noted that symbols could also be loaded directly into CPU registers if 8 or 32 bits per symbol were used. However, at 8 bits per symbol, the Re-Pair algorithm cannot create any additional symbols beyond the original set of terminals. 32 bits per symbol would also not be a practical choice, as the Re-Pair algorithm never creates an amount of new unique symbols that would require 32 bits to represent, therefore, using 32 bits per symbol would cause a significant loss in compression rates. This can be seen in Fig. 4. At the biggest evaluated block size (128 k), 19 bits per symbol are required at most. At block size 4 K, however, the number of bits per symbol never exceeds 16, which means that no compression is lost due to the aforementioned effect (1) at that block size. Since the average number of bits per symbol is not below 14 for any of the datasets at that block size, compressed size will also not increase significantly due to the aforementioned effect (2). Therefore, choosing $B = 4\,\text{k}$ or an equivalent s as the block size for BRPFC makes it possible to minimize the adverse effects on compression ratios and reap this change's benefits of lower dictionary access times, which will be shown in Sect. 5.

AVX-512 implementation

Similar to the scalar code in Algorithm 1, the vectorized version of EXPANDSYMBOL—EXPAND16SYMBOLS—consists of 5 main parts which we explain in the following. An overview of the parts is shown in Listing 1. In the first part, initialization of the AVX-512 registers necessary for the following parts is done. This is followed by the main loop that traverses the grammar tree, processing one node per vector element per iteration. The loop consists of three middle parts, one for processing vector elements that are non-terminals (branch 1), the second one for processing terminal vector elements (branch 2), and the third part for recombining the results from the two prior parts in preparation for the next iteration. This loop has a fixed number of 15 iterations, which is enough to expand symbols that expand to

```
// stack
const uint32_t SH = 10;
uint32_t stack[SH] = { 0 };
uint32_t stack_pos = 2;

// state
uint16_t current = symbol;

// output offset
uint16_t pos = 0;

// utility
uint16_t terminals = this->terminals
    ;

uint32_t mask =
    ~(uint32_t)0 >> (32 - G->numbits
    );
// return if outside of
//  the expected range
if (current >= this->rules +
    terminals)
    return 0;


```

Listing 2 Initialization in EXPANDSYMBOL

at most 8 characters, as discussed previously. The last part of EXPAND16SYMBOLS is the result output, where the output string, consisting of the concatenation of the expansions of all 16 symbols, is copied into the output buffer `str` and its length is returned. In the following sections, we will explain each part of EXPAND16SYMBOLS in more details. For each part, we show an equivalent scalar version—i.e., one that only expands a single symbol at a time—alongside the vectorized code. Unless otherwise noted, the scalar version performs the exact operations that the vectorized version does on a single symbol, while the vectorized version works on 16 symbols simultaneously.

Initialization: Listings 2 and 3 show the initialization phase of symbol expansion. In EXPAND16SYMBOLS, we zero-initialize a stack that is 16 times the stack height of the scalar version, since each of the 16 symbols we are expanding requires its own region on the stack. We fix the stack height to 10 per traversal. Since the expansion of symbols is limited to 8 characters, the stack height could actually be 8. However, to simplify branch 1 and 2, two additional elements are added at the bottom of the stack to make reading from and writing to the stack safe without bounds checking for symbols that have finished expanding. The stack position is initialized to 2 for all elements. This way, a stack position of 1 can signal that a symbol has been fully expanded. Once this has happened, the stack position for this symbol is not changed in subsequent iterations. Because of that, we can still read from the stack in each iteration and for each symbol at `stack_pos`—1 with-

```
1  // stack
2  const uint32_t SH = 10;
3  uint32_t stack[SH * 16] = { 0 };
4  __m512i stack_pos =
       _mm512_set1_epi32(2);
5  __m512i stack_offsets =
       _mm512_set_epi32(
6       SH * 0, SH * 1, SH * 2, SH * 3,
7       SH * 4, SH * 5, SH * 6, SH * 7,
8       SH * 8, SH * 9, SH * 10, SH *
       11,
9       SH * 12, SH * 13, SH * 14, SH *
       15);
10 // state
11 __m512i current =
12      _mm512_load_epi32(symbols);
13 // output string
14 __m512i pos = _mm512_setzero_epi32()
       ;
15 __m512i str0 = _mm512_setzero_epi32
       ();
16 __m512i str1 = _mm512_setzero_epi32
       ();
17 // utility
18 __m512i terminals =
       _mm512_set1_epi32(
19      this->terminals);
20 __m512i one = _mm512_set1_epi32(1);
21 __m512i mask = _mm512_set1_epi32(
22      ~(uint32_t)0 >> (32 - G->numbits
       ));
23 // set symbols to 0 that are outside
        of
24 //   the expected range
25 current = _mm512_mask_set1_epi32(
       current,
26      _mm512_cmpge_epi32_mask(current,
27           _mm512_set1_epi32(this->
       rules
28                + this->terminals)), 0);
```

Listing 3 Initialization in Expand16Symbols

```
1  uint16_t position = current -
2      terminal_count;
3  position = position < current
4      ? position : 0;
5
6  uint32_t child_rules = grammar[
       position];
7
8  uint16_t current1 = child_rules &
       mask;
9
10 uint16_t stack_value = child_rules >
        16;
11
12 stack[stack_pos] = stack_value;
13
14
```

Listing 4 Branch 1 in ExpandSymbol

out violating memory bounds. A `stack_offsets` register stores the offsets to the individual regions of the stack that are each used for one of the 16 symbols being expanded. Thereafter, we initialize the `current` register with the current symbol in the symbol tree that will be expanded in the next loop iteration. This register is initialized by loading 16 32-bit integers from the `symbols` array, which is an input parameter that contains the symbols that ought to be expanded.

In line 14, the `pos` register, which stores the length of each expanded string, is initialized to zero. To store the expanded strings, two 512-bit registers—`str0` and `str1`—are used. This is because each symbol can potentially expand to up to 8 characters, i.e., 8 bytes. Therefore, a maximum of $8 \cdot 16 = 128$ bytes is necessary to store all expanded strings, resulting in the use of two 512-bit, i.e., 64-byte, registers. As the length of each of the expanded strings is not known before the expansion is finished, the expanded string for each symbol is written to fixed 8-byte sized regions in the string registers. `str0` is used for the expanded strings of the first 8 symbols, and `str1` is used for the expanded strings of the other 8 symbols. As the scalar version expands only a single symbol, its outputs can be written directly to an output buffer and do not need to be buffered separately. Therefore, `str0` and `str1` do not exist in Listing 2.

Next, starting from line 18, three utility registers are initialized: The `terminals` register is initialized to contain the number of terminals in each of its 16 elements. It is later used to create the predicate that replaces the if-statement from the iterative version of ExpandSymbol. The `one` register just contains ones and is used several times for comparisons and arithmetic. The `mask` register is used to mask symbols decoded from the grammar array, in order to obtain the required left and right child symbols. Therefore, each of its 16 elements is initialized to have the lower `numbits` bits set. `numbits` is the number of bits used per symbol in the grammar, which is always 16 in our case.

Lastly, we validate that all input symbols are in the expected range of $[0, n + N)$ (line 25). Symbols that are not in this range are set to 0. This avoids out of bounds accesses to the grammar array. The check is necessary, since less than 16 symbols might be left in the bucket that is being decompressed, hence, some trailing symbols in the batch of 16 symbols might be invalid. If the symbol is found to be out of bounds in the scalar version, the function returns right away.

Branch 1: Listing 4 and Listing 5 show the equivalent of the first branch of Algorithm 1. This part loads the child symbols of the symbols in `current` from the grammar array, storing the left ones into `current1` for later use and putting the right ones on top of the stack for each element. To do so, n is subtracted from all 16 symbols in `current` to

```
1  __m512i position = _mm512_sub_epi32(
2      current, terminals);
3  position = _mm512_maskz_mov_epi32(
4      _mm512_cmple_epu32_mask(position
       ,
5          current), position);
6  __m512i child_rules =
7      _mm512_i32gather_epi32(position,
8          (int*)grammar, 4);
9  __m512i current1 =
10     _mm512_and_epi32(child_rules,
       mask);
11 __m512i stack_values =
12     _mm512_srli_epi32(child_rules,
       16);
13 __m512i final_stack_offsets =
14     _mm512_add_epi32(stack_offsets,
15         stack_pos);
16 _mm512_i32scatter_epi32((int*)stack,
17     final_stack_offsets,
18     stack_values, 4);
```

Listing 5 Branch 1 in EXPAND16SYMBOLS

```
1
2  uint32_t current2 = stack[stack_pos
       - 1];
3
4
```

Listing 6 Branch 2 in EXPANDSYMBOL

```
1  final_stack_offsets =
       _mm512_sub_epi32(
2      final_stack_offsets, one);
3  __m512i current2 =
4      _mm512_i32gather_epi32(
5      final_stack_offsets, (int*)stack
       , 4);
```

Listing 7 Branch 2 in EXPAND16SYMBOLS

obtain the indices, stored in `position`, of the corresponding non-terminals in the grammar array. Since not all symbols in `current` are necessarily non-terminals, but can also be terminals, this subtraction can produce negative values, resulting in invalid indices. Because two-complement integers are used here, interpreting negative integers as unsigned ones results in values above $2^{31} - 1$. We utilize this by creating a mask that is set to one for elements that are in the expected range $[0, n + N)$. This mask is obtained by checking if elements are less than or equal to the elements in `current`. The conditional `move` operation zeroes the elements in `position` that this mask is set to zero for in line 3. This makes it possible to use these elements as an index into the grammar array safely. The fact that they have no meaningful value does not matter here, since the corresponding decoded symbols are discarded anyways at a later stage.

Thereafter, the child symbols are gathered directly from the grammar array using a single 32-bit gather in line 6. This is possible since the child symbols for a non-terminal are guaranteed to have a 32-bit offset in the grammar array and to be 32 bits in total size, thanks to fixing the size of symbols to 16 bits. Subsequently, decoding the left child symbols for `current1` is simply a matter of masking away the higher 16 bits of the gathered elements (line 9). Then, to store the right child symbols on the stack, we decode them by right-shifting the gathered elements by 16 bits (line 11). We compute the `stack` offsets by adding `stack_offsets` to `stack_pos` and storing that in `final_stack_offsets` in line 13. Since these do not point to contiguous memory locations, we use a `scatter` instead of a `store` operation to write the `stack_values` to the right locations.

Branch 2: Listing 6 and Listing 7 show the equivalent of the second branch of Algorithm 1. Branch 2 is responsible for fetching symbols from the stack to be possibly used as new current symbols in the next loop iteration. Symbols need to be fetched from `stack_pos`—1 (line 1), where `stack_pos` has already been computed previously. Once this is done, symbols are loaded into `current2` from the stack using a 32-bit `gather` operation in line 3. As the scatter from line 16 of branch 1 writes to different positions on the stack than the ones that the `gather` in line 3 of branch 2 reads from, the data fetched in branch 2 cannot have been overwritten by branch 1. Reading at the `stack_pos`—1 offsets is safe, as none of the elements in `stack_pos` can be less than one. Therefore, subtracting one from the previously calculated offsets always produces a safe offset that can be used to read without causing a memory protection violation.

Predicate evaluation: Listing 8 and 9 show how the previously computed results of branch 1 and branch 2 can be combined without branching using predicated operations. The main predicate is evaluated first, and the mask `predicate` is created from it in line 1. This is equivalent to the condition of the original if-statement in Algorithm 1. The created mask consists of ones for terminals and zeroes for non-terminals in `current`. The second mask stored in the register `not_done_predicate`, created in line 4, is required to keep track of which symbols have already been fully expanded. This state is indicated by `stack_pos == 1`. The mask is therefore obtained using an inequality comparison between `stack_pos` and `_1`.

Before `current` and `stack_pos` are updated, the terminals in `current` are extracted to become part of the expanded string of the respective symbol. This is done by masking `current` to obtain only the symbols that are terminals in `current_masked`. `current_masked` is then split up into the registers `str0_add` and `str1_add`, where

```
1  bool is_terminal =
2      current < terminal_count;
3
4  bool not_done_predicate = stack_pos
       != 1;
5
6  uint16_t current_masked =
       is_terminal
7      ? current : 0;
8
9  str[pos] = current_masked;
10 current = is_terminal
11     ? current2 : current1;
12 uint16_t stack_pos_tmp = is_terminal
13     ? stack_pos + 1 : stack_pos - 1;
14
15 stack_pos = not_done_predicate
16     ? stack_pos_tmp : 1;
17 uint16_t pos_add =
18     not_done_predicate &&
       is_terminal
19     ? 1 : 0;
20
21 pos += pos_add;
22 if (!not_done_predicate) break;
```

Listing 8 Combining branch results in ExpandSymbol

```
1  __mmask16 is_terminal =
2      _mm512_cmplt_epi32_mask(current,
3          terminals);
4  __mmask16 not_done_predicate =
5      _mm512_cmpneq_epi32_mask(
6          stack_pos, one);
7  __m512i current_masked =
8      _mm512_maskz_mov_epi32(
9          is_terminal, current);
10 __m512i str0_add =
       _mm512_cvtepi32_epi64(
11     _mm512_extracti32x8_epi32(
12         current_masked, 0));
13 __m512i str1_add =
       _mm512_cvtepi32_epi64(
14     _mm512_extracti32x8_epi32(
15         current_masked, 1));
16 str0_add = _mm512_sllv_epi64(
       str0_add,
17     _mm512_cvtepi32_epi64(
18     _mm512_extracti32x8_epi32(pos,
       0)));
19 str1_add = _mm512_sllv_epi64(
       str1_add,
20     _mm512_cvtepi32_epi64(
21     _mm512_extracti32x8_epi32(pos,
       1)));
22 str0 = _mm512_or_si512(str0,
       str0_add);
23 str1 = _mm512_or_si512(str1,
       str1_add);
24 current = _mm512_mask_mov_epi32(
       current1,
25     is_terminal, current2);
26 __m512i stack_pos_tmp =
27     _mm512_mask_mov_epi32(
28         _mm512_add_epi32(stack_pos,
       one),
29         is_terminal,
       _mm512_sub_epi32(
30             stack_pos, one));
31 stack_pos = _mm512_mask_mov_epi32(
       one,
32     not_done_predicate,
       stack_pos_tmp);
33 __m512i pos_add =
       _mm512_maskz_mov_epi32(
34     not_done_predicate,
35     _mm512_maskz_mov_epi32(
       is_terminal,
36         _mm512_set1_epi32(8)));
37 pos = _mm512_add_epi32(pos, pos_add)
       ;
38 if (not_done_predicate == 0) break;
```

Listing 9 Combining branch results in Expand16Symbols

the first stores the symbols being added to the first 8 expanded strings and the latter those being added to the second 8 expanded strings. Thereby each new symbol has a 64-bit slot in one of the registers. As the final expanded strings also take up at most 64 bits, this makes it possible to shift the new terminals in `str0_add` and `str1_add` to their final positions in the respective string using the information about the current output expanded lengths from the `pos` register (lines 16–21). Once the terminals are at the right position in their 64-bit slots, they can be combined into the string registers using an or-instruction. Lastly, `current`, `stack_pos`, and `pos` are updated to reflect the changes from the current iteration and prepare for the next iteration. Using a conditional `move` operation with the `predicate` mask, elements in `current` are updated to either become elements from `current1` or `current2` if the corresponding element in `current` was a non-terminal or a terminal, respectively (cf. line 24). Updating `stack_pos` takes two conditional `move` operations: The first one in line 26 increments elements if the respective element in `current` was a non-terminal and decrements them if it was a terminal. The second operation in line 31 sets elements to one if the respective symbol has already been fully expanded to ensure that no elements of `stack_pos` drop below one. `pos` is updated using a similar logic in lines 33 and 37: an offset (8) is added only to elements that have not already been fully expanded and where the respective element in `current` was a terminal. Finally, line 38 serves as an early exit from the symbol expansion loop if all 16 symbols are already fully expanded.

Result output: After the main loop of Expand16Symbols has finished, the resulting 16 individual expanded strings are processed to form a single contiguous string that is the expansion of all 16 symbols. The code for this is shown in Listing 10. We do not show equivalent scalar code here, as

```
1  pos = _mm512_srli_epi32(pos, 3);
2  uint8_t expanded_chars[16];
3  _mm_store_si128(reinterpret_cast<__m128i*>(expanded_chars),
       _mm512_cvtepi32_epi8(pos));
4  _mm512_storeu_si512(str, str0);
5  _mm512_storeu_si512(str + 64, str1);
6
7  uint32_t offset = expanded_chars[0];
8  for (size_t i = 1; i < 16; i++) {
9      memmove(str + offset, str + i * 8, 8);
10     offset += expanded_chars[i];
11 }
12
13 return offset;
```

Listing 10 Result output in EXPAND16SYMBOLS.

the scalar code outputs directly to a buffer and only needs to return the length of the expanded string at this point.

All elements in `pos` are first divided by 8 with a left shift operation in line 1. After this, they represent byte counts as opposed to the bit counts they represented earlier. These byte counts of the number of expanded characters for each original symbol are then stored in `expanded_chars` as a local buffer (in line 3) for use in the scalar code that concatenates the expanded strings. The expanded strings themselves are stored in the output buffer `str` in lines 4 and 5. Following this, the individual expanded strings are concatenated by moving each string directly behind its predecessor using successive calls to `memmove` in the loop from lines 8–11. Since each call to `memmove` moves a fixed number of 8 bytes, these calls are optimized by the compiler to become simple loads and stores without the cost of a function call. The total number of expanded characters is kept in the `offset` variable and is finally returned as the result of EXPAND16SYMBOLS in line 13.

We note that it may also be possible to implement the concatenation of the individual output strings entirely vectorized by using `compress_epi8` instructions. However, the `compress` instructions operating on byte granularity are part of the VBMI2 extension of AVX-512, which was not available to us at the time of implementing this, hence we chose a scalar implementation here.

AVX-512 optimizations

While developing EXPAND16SYMBOLS, several opportunities for further optimization were considered as well, but all of the following ideas resulted in no performance improvements or even slowdowns, so they are not included in our final proposed version.

Merging gathers: As gathers and scatters are the most expensive operations in AVX-512, it is desirable to avoid them as much as possible. Two gathers are used in the final version of EXPAND16SYMBOLS: One for gathering child symbols from the grammar array, and one for gathering possible new current symbols from the stack. Ultimately, the resulting elements of these gathers are used fully exclusively when updating `current`. It is therefore possible to merge the two gathers into a single one. To achieve this, the stack cannot be a freely allocated buffer anymore, but is instead kept in front of the grammar array. This way, a single gather can be used to access the grammar array with positive offsets, and the stack with negative offsets. Of course, the offsets have to be properly prepared for that. Applying this in practice, we saw a slight performance decrease with this change instead of the expected improvement from removing one gather. This is because the *critical path* of the final version of EXPAND16SYMBOLS is formed by the gather from the grammar array. The gather from the stack is insignificant in comparison, because the stack can always be kept in L1 cache, as opposed to the grammar array, which does not fully fit in L1 cache. By adding more operations to calculate the modified offsets for the merged gather, the length of the critical path is thus increased, slowing down the execution of EXPAND16SYMBOLS.

Masked gathers/scatters: As all AVX-512 operations support masking away individual elements, it could be beneficial to mask elements in the scatter and gather operations in EXPAND16SYMBOLS, to avoid unnecessary memory accesses. This can be done without problems, as the masks required for this, namely `not_done_predicate` and `predicate`, can also be generated at the beginning of each iteration. However, we found that this does not make a difference in performance in our case and therefore omitted this change in favor of simplicity.

Stack in registers: Lastly, we explored keeping the stack in AVX-512 registers instead of in memory, in order to avoid the expensive gather and scatter operations used to read and write from it. To do this, 8 registers are necessary, as each register

Table 2 Datasets used in our experiments

	# Strings (Mio)	Size (MB)	⌀ String size (B)
UK	18.52	1438.71	76.68
INDOCHINA	7.41	642.71	85.68
ENWIKI	35.28	708.34	19.08
GEONAMES	7.31	114.59	14.68

can hold 16 elements. With this change, the stack can have a height of 8, since unsafe elements in `stack_pos` are handled implicitly. To read and write from this stack, masks have to be computed, indicating for each of the 8 layers, whether elements should be written to or read from the layer. The reading and writing can then be done using conditional move operations that merge the elements in two registers based on a mask. However, as this requires an extensive amount of operations proportional to the height of the stack, the version with the memory-based stack outperforms this version and we therefore dismiss this change.

5 Experimental evaluation

In this section, we present several experiments that prove the effectiveness of the compression and decompression techniques proposed in Sects. 3 and 4. We evaluate RPFC compression with the modifications that make improved access times possible, namely limiting non-terminals to expand to at most 8 terminals and using exactly 16 bits for all symbols. The newly proposed BRPFC and sampled BRPFC methods are evaluated as well, followed by an evaluation of decompression performance, and an analysis of the sensitivity of decompression performance regarding the available cache size. Additionally, we present results from evaluating both compression and decompression in the real-world context of an IMDBMS.

We use the bucket size $b = 16$ for front coding in all of our experiments, as this has proven to be a good tradeoff between compression and dictionary access times in practice. As a baseline, we use the original RPFC method by Navarro et al. [7].

Experimental setup

We run the experiments on a system with Intel® Xeon® Platinum 8180 CPUs [26] with 1 MiB/core of L2-cache, 38,5 MiB 11-way associative non-inclusive L3-cache, 183 GiB RAM, and CentOS 7.5.1804. If not mentioned otherwise, we run the experiments single-threaded. The code was compiled using gcc 7.4.0 with -O9 optimizations.

We use the following datasets in our experiments:

- UK: URLs obtained from a 2002 crawl [27] of the ".uk" domain using the UbiCrawler [28].
- INDOCHINA: URLs collected from a 2004 Indochina domain crawl [27].
- ENWIKI: Page titles from English Wikipedia of Dec. 2018 [29].
- GEONAMES: *asciiname* column of the Geonames geographical name dump [30].

Table 2 shows relevant statistics of the datasets. They are selected to cover common use cases for string columns, such as natural language (ENWIKI), names (GEONAMES), and machine-readable data (INDOCHINA, UK). They also range from short average string size (GEONAMES, ENWIKI) to long average string size (INDOCHINA, UK). The UK and an older version of the ENWIKI datasets were used in [5–7,10]. All 4 datasets were also used by Kanda et al. [19].

Compression ratios

In this section, we compare the compression ratios of the following string dictionary compression methods:

- RPFC: The original Re-Pair front coding by Navarro et al. [7].
- RPFC8: RPFC expanding to at most 8 terminals as described in Sect. 4.
- RPFC8 16bps: RPFC expanding to at most 8 terminals and using 16 bits per symbol (bps) as described in Sect. 4.
- BRPFC: the block-based variant of RPFC as introduced in Sect. 3.
- Sampled BRPFC: the sampled variant of BRPFC as introduced in Sect. 3.
- PFC: Plain front coding as implemented by Navarro et al. [7].

BRPFC and sampled BRPFC both include the two changes facilitating fast vectorized decompression, as RPFC8 16bps does. However, we chose to use the fixed number of 16 bits per symbol only in the grammars of the compressed dictionaries for those two methods. The compressed texts themselves are encoded using as many bits per symbol as are required based on the total number of distinct symbols per block. Additionally for BRPFC and sampled BRPFC, strings are stored prefixed with their length instead of zero-terminated. This allows storing binary strings that may contain zero bytes, but affects compression quality negatively. The aforementioned two changes are in contrast to our previous BRPFC results [11]. We also compare against two general-purpose compressors, gzip (at the highest compression level 9), and Snappy [31]. The reported compression times and ratios are the result of compressing the datasets with the respective compression utility without any further

pre- or postprocessing. General-purpose compression cannot be used for string dictionaries, as it does not support random accesses, but it is nevertheless worthy comparing against, as it illustrates which compression could optimally be obtained on a given dataset. As Snappy is only available as a library, we use the snzip command line utility [32] to run Snappy compression on our input files.

Table 3 shows the measured compression ratios along with the respective time taken for compression, which will be discussed in the following section. Compression ratios are obtained by dividing the compressed size of the string dictionary by the uncompressed size of the dataset. We experiment with different BRPFC block sizes s ranging from 64 thousand to 64 million symbols per block. In combination with the compression time results, this allows us to determine the block size with the best trade-off between compression time and compression ratio.

For RPFC, the results show that the 8 terminal limitation (RPFC8) increases compression ratios. This increase is, however, drastically different between datasets, as the compression ratio for INDOCHINA increases by 50%, while the one for UK only increases by 30%, and the ratios for ENWIKI and GEONAMES increase by less than 10%. However, the compressed size of all datasets stays below 67% of the compressed size achieved by plain front coding. Compression ratios are *improved* again by using a fixed number of 16 bits per symbol. This is in contrast to our forecast from Sect. 4, where we predicted that this change would *deteriorate* compression quality. The effect is an artifact of the bit-packed encoding of the compressed blocks used here. Because of this encoding, adding Re-Pair rules can inflate the compressed size of the dictionary, even though the number of symbols is reduced, due to the fact that adding a new rule can cause more bits to be used for *all* symbols in the final encoded compressed text. This has also been observed by Yoshida and Kida [33] when using a similar variable-length coding for Re-Pair compression. Here, when limiting symbols to use 16 bits, Re-Pair rule creation is also limited to rules that can be represented within this bitwidth, and hence the negative effect of adding rules that deteriorate compression quality is weakened. Overall, the compressed size of all datasets when using RPFC8 16 bps is below 63% of the compressed size when using plain front coding.

For BRPFC, we observe in general that compression ratios improve, or at least stay similar with increased block sizes. For the datasets with relatively small average string sizes—ENWIKI and GEONAMES—there is a general positive trend of improved compression ratios at higher block sizes. With the two datasets that feature relatively large average string sizes, UK and INDOCHINA, there is a slight negative trend. This is expected as larger block sizes allow Re-Pair to extract more redundancy from its input data. At the same time, however, if the Re-Pair algorithm creates more rules, an increasing number of bits is required to store each symbol in the compressed texts. This negative effect is more apparent for the datasets with larger average string sizes and slightly outweighs the positive effect of being able to exploit more redundancy. Because RPFC8 16bps always uses 16 bits per symbol, even if they are not required, it is outperformed by BRPFC on UK and INDOCHINA, especially at small block sizes. Contrary to that BRPFC compression ratios for ENWIKI and GEONAMES only come close to RPFC8 16bps at larger block sizes. This can be attributed to the reduced ability of Re-Pair to exploit redundancy at small block sizes, as well as to the previously mentioned change in BRPFC that allows storing binary strings at the cost of decreased compression quality. In summary, we observe that the block size where BRPFC compresses better than RPFC8 16bps is lower for the datasets with longer average string sizes (UK and INDOCHINA), and non-existent for the ones with smaller average string sizes (ENWIKI and GEONAMES). Compared to front coding (PFC), BRPFC still significantly improves compression at all block sizes. Compressed sizes are reduced by at least 24% at the lowest evaluated block size, and more than that at other block sizes.

For sampled BRPFC, we evaluate the resulting compression ratios at varying superblock sizes S ranging from 64 thousand symbols to 128 million symbols. We use a block size of $s = 8\,\text{M}$. It should be noted, however, that sampled BRPFC's compression quality is not sensitive toward the block size s, but only toward the superblock size S. The results are shown in Table 4. As expected, compression ratios improve with increased superblock sizes because the quality of the shared grammar improves with an increased sample size. However, the returns of increasing the superblock size start diminishing after $S = 8\,\text{M}$. At this point, the compression ratios can be compared to those of BRPFC shown in Table 3. For UK and INDOCHINA, sampled BRPFC cannot quite reach the compression ratios of BRPFC, but the difference is within 2% for most block sizes s. For ENWIKI, sampled BRPFC is better than BRPFC for block sizes $s < 8\text{M}$ and for GEONAMES it is even strictly better than BRPFC. Sampled BRPFC benefits from only storing a single shared grammar for the whole dictionary here, instead of storing a grammar for each block, potentially partially redundant to other grammars in the dictionary. As BRPFC does, sampled BRPFC also significantly improves compression over PFC, reducing compressed sizes by at least 35%.

Comparing to the general-purpose compressors, one can observe that the requirement of supporting random accesses to strings severely limits compression quality, as gzip compresses up to 3× stronger than BRPFC and sampled BRPFC.

To confirm the effectiveness of sampled BRPFC's sampling scheme, we perform a brief analysis of the Re-Pair grammars created by sampled BRPFC. The results are shown in Table 5. The table shows the number of rules in the shared

Table 3 Compression ratios and compression times of the evaluated string dictionary compression methods and two general-purpose compressors

		UK		INDOCHINA		ENWIKI		GEONAMES	
RPFC [7]		14.2%	1544.9 s	12.0%	358.1 s	30.8%	984.7 s	32.1%	53.1 s
RPFC8		18.3%	1140.4 s	18.0%	509.2 s	33.8%	870.8 s	35.4%	52.4 s
RPFC8 16 bps		17.4%	813.9 s	16.8%	216.4 s	31.4%	604.5 s	32.3%	41.9 s
BRPFC varying s	s=64 k	16.7%	51.4 s	15.0%	20.9 s	38.2%	64.8 s	42.4%	11.0 s
	s=128 k	16.6%	53.7 s	15.0%	20.0 s	37.5%	65.7 s	40.8%	12.0 s
	s=256 k	16.7%	57.9 s	15.2%	21.4 s	35.6%	70.6 s	39.6%	13.0 s
	s=512 k	16.8%	60.2 s	15.4%	21.8 s	35.2%	81.6 s	39.3%	13.8 s
	s=1 M	16.9%	66.4 s	15.7%	23.0 s	34.9%	82.0 s	39.0%	15.1 s
	s=2 M	16.8%	74.6 s	16.0%	25.1 s	34.5%	91.1 s	38.5%	17.1 s
	s=4 M	16.8%	93.3 s	16.3%	29.0 s	33.2%	130.9 s	36.8%	22.2 s
	s=8 M	16.6%	90.9 s	16.4%	35.4 s	32.5%	112.5 s	35.9%	21.5 s
	s=16 M	16.7%	83.8 s	16.4%	33.7 s	32.3%	127.2 s	35.5%	20.7 s
	s=32 M	16.9%	91.1 s	16.6%	31.4 s	32.2%	125.6 s	35.2%	21.6 s
	s=64 M	17.2%	77.9 s	16.8%	30.4 s	32.2%	102.5 s	35.1%	18.4 s
Sampled BRPFC		17.8%	8.03 s	17.3%	4.45 s	33.0%	9.87 s	35.1%	4.39 s
PFC [7]		27.5%	3.5 s	29.9%	1.5 s	50.4%	3.3 s	56.1%	0.5 s
gzip (-9)		7.3%	45.2 s	5.9%	21.7 s	26.1%	127.8 s	29.1%	22.8 s
Snappy (framing2 format) [32]		13.2%	5.6 s	11.8%	2.2 s	41.3%	8.1 s	46.6%	1.5 s

The BRPFC block size s refers to the number of symbols in each block before the blocks are Re-Pair compressed. Sampled BRPFC uses the parameters $s = 8\,\text{M}$ and $S = 8\,\text{M}$

Table 4 Compression ratios and compression times of sampled BRPFC with varying superblock size S

	UK		INDOCHINA		ENWIKI		GEONAMES	
S=64 k	22.7%	5.23 s (0.01 s)	22.8%	2.80 s (0.01 s)	39.3%	5.58 s (0.01 s)	41.3%	1.02 s (0.01 s)
S=128 k	22.4%	6.73 s (0.03 s)	20.4%	2.16 s (0.03 s)	38.5%	5.88 s (0.03 s)	38.5%	1.05 s (0.03 s)
S=256 k	22.0%	5.38 s (0.06 s)	19.8%	2.25 s (0.05 s)	35.9%	5.89 s (0.06 s)	38.2%	1.18 s (0.06 s)
S=512 k	20.3%	6.88 s (0.14 s)	19.2%	2.26 s (0.11 s)	35.6%	6.08 s (0.13 s)	37.9%	1.23 s (0.13 s)
S=1 M	19.7%	5.69 s (0.32 s)	18.7%	2.42 s (0.23 s)	35.4%	6.52 s (0.31 s)	37.6%	1.43 s (0.27 s)
S=2 M	19.2%	5.85 s (0.72 s)	17.8%	3.29 s (0.53 s)	35.0%	8.66 s (0.75 s)	37.2%	1.78 s (0.65 s)
S=4 M	18.2%	8.67 s (1.75 s)	18.0%	3.30 s (1.16 s)	33.4%	10.12 s (2.15 s)	35.5%	2.89 s (1.62 s)
S=8 M	17.8%	8.03 s (2.81 s)	17.3%	4.45 s (2.30 s)	33.0%	9.87 s (3.20 s)	35.1%	4.39 s (3.13 s)
S=16 M	17.7%	10.48 s (5.15 s)	17.2%	6.21 s (4.06 s)	32.9%	12.19 s (5.76 s)	34.9%	7.67 s (6.18 s)
S=32 M	17.6%	18.32 s (11.11 s)	17.1%	9.85 s (7.56 s)	32.7%	17.56 s (10.86 s)	34.8%	13.80 s (11.93 s)
S=64 M	17.6%	23.51 s (17.67 s)	17.1%	16.60 s (14.09 s)	32.7%	29.36 s (21.74 s)	34.8%	26.94 s (24.56 s)
S=128 M	17.6%	40.01 s (33.52 s)	17.1%	29.74 s (26.94 s)	32.7%	50.06 s (40.95 s)	34.8%	25.48 s (23.13 s)

The time shown in parentheses is the time taken to compress only superblock, while the other value is the full compression time

Table 5 Sampled BRPFC grammars (superblock size $S = 8\,\text{M}$)

	# Rules	Overlap with BRPFC ($s = 8\,\text{M}$) (%)
UK	65,278	84.1
INDOCHINA	65,278	74.8
ENWIKI	65,278	98.2
GEONAMES	65,278	94.9

grammars of sampled BRPFC dictionaries in the first column. The second column shows the percentage of these rules that are also present in the BRPFC dictionaries of the respective dataset at the same block size as the superblock size used for sampled BRPFC. It can be observed that 65,278 rules are created for all datasets, which is the maximum number of rules that can be created given our limitation of using at most 16 bits per symbol. Additionally, all datasets exhibit high percentages of 74.8–98.2% of rule overlap between sampled BRPFC and BRPFC. The facts that the sampled

BRPFC grammars utilize the maximum possible number of rules to achieve compression and that at least 74.8% of these rules can also be found in the corresponding BRPFC grammars, show that sampled BRPFC is effective at replicating the compression performed by BRPFC at a lower computational cost. Nevertheless, there may be cases where sampled BRPFC fails to produce results as good as BRPFC that are not captured with the datasets evaluated here. For example, if the strings in different blocks of the dictionary use entirely different character sets—e.g., the first half of the dictionary is alphanumeric, and the second half consists solely of special characters—then it would be difficult for a shared grammar to achieve the same compression quality as separate grammars for these very different blocks could. However, to the best of our knowledge, such extreme cases are uncommon in practice and we did not encounter such issues in our testing. We therefore believe sampled BRPFC to be applicable in the majority of cases. If unfavorable input data are encountered in practice, it is also still possible to fall back to BRPFC to preserve compression quality at the cost of higher compression times.

Compression times

In this section, we compare the time necessary for dictionary construction with each of the methods evaluated in the previous sub-section. The results are shown alongside the compression ratios in Table 3. More detailed results for sampled BRPFC at varying superblock sizes S are presented in Table 4. Looking at the RPFC results, we observe that RPFC8 16 bps compression times are faster than those of RPFC. This is because the number of symbols is limited by their 16-bit representation in RPFC8 16 bps, thereby reducing the number of iterations done by the Re-Pair algorithm. Additionally, the number of rules that are created by the Re-Pair algorithm is reduced by the limitation of rules expanding to at most 8 symbols, which also reduces the iteration count.

The newly proposed BRPFC compression is roughly an order of magnitude faster than that of RPFC at smaller block sizes and also stays faster even at larger block sizes. BRPFC compression times increase with increasing block sizes.

Varying the superblock size S for sampled BRPFC as shown in Table 4, we observe that increasing the superblock size results in a roughly proportional increase in superblock compression time (shown in parentheses). The remaining part of compression time, which is primarily comprised of the time to run the recompression, stays mostly constant with increasing S. It does slightly go up with increasing S because that also results in a more complex shared grammar and therefore a longer runtime of the recompression algorithm, but that effect is relatively small in comparison with the cost of compressing the superblock itself. Note that the proportional increase in superblock compression time does not hold true for GEONAMES going from $S = 64\,\text{M}$ to $S = 128\,\text{M}$. This is because the dataset consists of only 64.3 M symbols after front coding. Consequently, no more symbols than that can be sampled, and the target size of $S = 128\,\text{M}$ cannot be reached, resulting in an actual superblock size of 64.3 M symbols even at $S = 128\,\text{M}$.

Comparing sampled BRPFC at $S = 8\,\text{M}$ to the other compression methods shown in Table 3, it becomes apparent that the compression times of BRPFC are further reduced with the sampled BRPFC variant, which is up to two orders of magnitude faster than the original RPFC. As noted in the previous section, sampled BRPFC still significantly improves compression over front coding (PFC) while at the same time achieving the aforementioned speedup compared to RPFC.

In light of sampled BRPFC's comparatively very fast compression times, it should be noted that BRPFC may still yield better compression quality for certain datasets at acceptable compression times. Therefore, depending on the exact use case, one can choose between BRPFC and sampled BRPFC to get optimal results. In comparison with the version of BRPFC presented in our previous work [11], it is not as clear any more which block size s is optimal to use with BRPFC. This is because we encode symbols with variable bitwidths to improve compression ratios at smaller block sizes. In particular, the compression ratios of BRPFC have become less sensitive toward the chosen block size.

In summary, both BRPFC and sampled BRPFC significantly reduce compression times in comparison with RPFC, while mostly preserving RPFC's compression quality. Sampled BRPFC reaches the faster compression times of the two methods and comes very close to plain front coding, our baseline. It is only between 2.5 and 9 times slower than PFC on the evaluated datasets while reducing the compressed size to as low as 58% of the compressed size reached with PFC.

In terms of compression times, BRPFC performs similarly in comparison with gzip, while sampled BRPFC compresses significantly faster than gzip across all datasets. The more lightweight Snappy compressor features compression times slightly faster than sampled BRPFC, while compressing UK and INDOCHINA much better than sampled BRPFC, but compressing ENWIKI and GEONAMES worse than sampled BRPFC.

Access times of re-pair front coding

In this section, we compare the access times of the compressed dictionaries. As discussed in Sect. 2, the access operations supported by string dictionaries are *extract*(*id*) and *locate*(*str*), which return a string given a value identifier *id* or return the value identifier of the string *str*, if *str* is present in the dictionary, respectively. We evaluate these operations separately by executing each operation 10^6 times

Table 6 RPFC *extract* (top) and *locate* (bottom) times (μs)

Extract	RPFC Orig.	RPFC Iter.	RPFC AVX-512	RPFC 16 bps AVX-512	BRPFC 16bps AVX-512	Sampled BRPFC 16 bps AVX-512	PFC
UK	2.47	1.99	1.16	0.85	1.07	1.02	0.42
INDOCHINA	2.53	1.87	1.20	0.90	1.00	0.97	0.38
ENWIKI	1.66	1.34	0.86	0.61	0.86	0.78	0.37
GEONAMES	1.28	0.98	0.70	0.51	0.62	0.58	0.31
Locate	RPFC Orig.	RPFC Iter.	RPFC AVX-512	RPFC 16 bps AVX-512	BRPFC 16 bps AVX-512	Sampled BRPFC 16 bps AVX-512	PFC
UK	4.15	3.53	2.50	1.92	2.30	2.17	1.46
INDOCHINA	3.77	3.13	2.23	1.86	2.03	2.03	1.34
ENWIKI	2.75	2.38	1.71	1.31	1.67	1.56	1.09
GEONAMES	1.87	1.57	1.25	1.02	1.17	1.14	0.83

using a pre-generated set of random predicates and repeating that measurement 10 times.

We report the average execution time of a single access operation in Table 6. **RPFC Orig.** represents the runtimes of the original implementation of RPFC from [7]. **RPFC Iter.** shows the runtimes of the optimized iterative version presented in Sect. 4. **RPFC AVX-512** refers to an implementation using the vectorized version of Re-Pair symbol expansion that operates on **RPFC8** compressed dictionaries only. Finally, **RPFC 16bps AVX-512** uses vectorized symbol expansion with the simplifications made possible by storing 16 bits per symbol. Additionally, we report in the **BRPFC 16 bps AVX-512** column the operation runtimes for a BRPFC8 16 bps compressed dictionary with $s = 8$ M using vectorization and in **Sampled BRPFC 16 bps AVX-512** the equivalent values for a dictionary compressed with sampled BRPFC at $S = 8$ M. For comparison, we also report the access operation runtimes of PFC.

The results show that the optimized iterative version already improves upon the original implementation, resulting in a speedup of at least 24% and 15% for *extract* and *locate* operations, respectively, on all datasets, without requiring any changes to the original compressed dictionary structure. If we accept to reduce compression rates over the original RPFC method, then another drastic speedup of at least 40% and 26% for *extract* and *locate*, respectively, is achieved by vectorizing Re-Pair expansion. Finally, avoiding the expensive decoding of bit-packed arrays by fixing the number of bits per symbol at 16 improves access times significantly. As this change even improves compression ratios over just limiting symbol expansion to 8 terminals, it is clearly superior to its predecessor. It improves *extract* and *locate* times by 2.5–2.9 and 1.8–2.2 times, respectively, over the original RPFC implementation. Of course, accessing front-coded dictionaries is still faster, as the additional cost of Re-Pair cannot be entirely avoided. The **16bps AVX-512** method makes *extract* and *locate* 1.6–2.4 and 1.2–1.4 times slower than those for PFC, respectively. This can be an acceptable trade-off given the drastically improved compression ratios over front coding in scenarios where reducing main memory load is desirable. Unfortunately, the access times for BRPFC compressed dictionaries shown in column **BRPFC 16 bps AVX-512** are worse than for RPFC compressed ones when AVX-512 decompression is used. This is because BRPFC uses one grammar for each block, instead of one for the whole dictionary. This results in increased numbers of cache misses when accessing the compressed dictionary at random locations. The block-based nature of BRPFC dictionaries also results in an additional indirection layer in comparison with RPFC, producing more cache misses. These effects cannot be avoided without going back to using a single grammar for the whole dictionary. The order of the dictionary is also required to be lexicographical by front coding. This prohibits ordering strings in the dictionary differently to improve data locality for certain access patterns. Sampled BRPFC—shown in column **Sampled BRPFC 16bps AVX-512**—slightly improves the access times over full BRPFC, because only a single grammar is used there. However, it is still beaten by **RPFC 16bps AVX-512** due to the disadvantages of the block-based dictionary structure.

BRPFC's *extract* and *locate* operations are 2.0–2.6 and 1.4–1.6 times slower than those of PFC, respectively. Sampled BRPFC performs slightly better and is 1.9–2.5 times slower than PFC for *extract* and 1.4–1.5 times slower for *locate* operations. We argue that these times are still reasonable given the improved compression over PFC and the drastically reduced compression times over RPFC.

Cache sensitivity analysis

In Sect. 4, we argued that the performance of the lookup operations in the compressed dictionary depend on the latencies

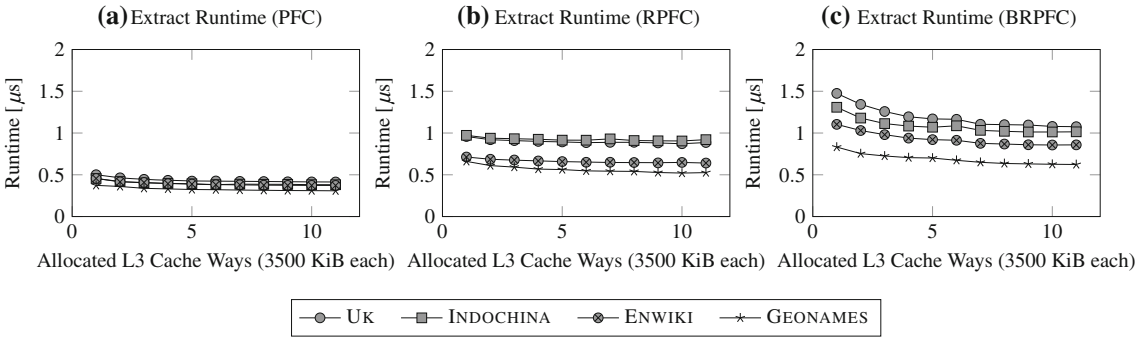

Fig. 10 L3 Cache sensitivity for extract

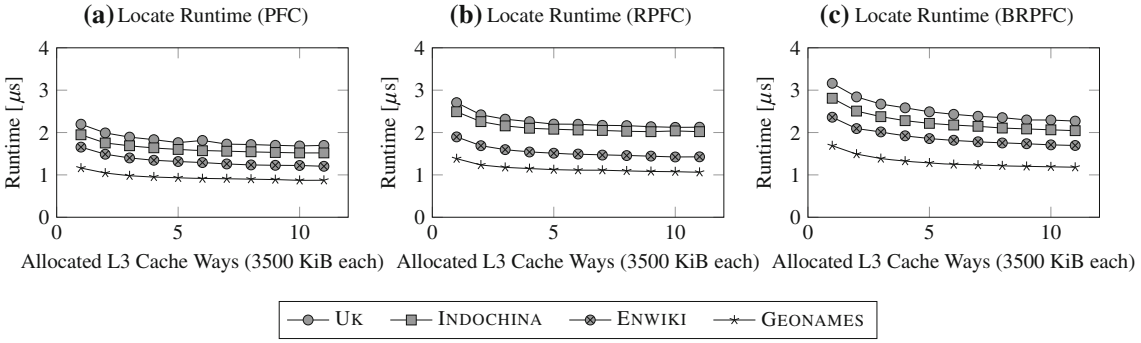

Fig. 11 L3 Cache sensitivity for locate

of memory accesses. In this section, we analyze how sensitive PFC, RPFC, and BRPFC are regarding the size of the last/3rd level cache (LLC) for the *extract* operation in Fig. 10 and the locate operation in Fig. 11. To analyze this, we rely on the cache allocation technology (CAT) of modern Intel CPUs [34] that can restrict data allocation originated from certain processor cores to portions (ways) of the LLC. By forcing our test process to run on a core with such restricted access, we emulate reduced L3 cache availability. As the LLC of the Intel processor used in our experiments is 11-way associative with 38.5 MB total capacity, we can assign cache in portions of 3.5 MB shown on the x-axis of these figures. For these operations, we report the response times for the *extract* and *locate* operations. As with the previous experiments on access times, the reported times refer to a single operation. They were obtained by executing 10^6 operations with a pre-generated set of predicates 10 times and calculating the average time for a single operation from that. We run the BRPFC experiments on dictionaries with block size $s = 8$ M. Note that the results using the full cache correspond to the measurements we report in Table 6. As we discussed, PFC generally has lower response times than the Re-Pair methods due to the additional Re-Pair decompression overhead imposed by those. Comparing Fig. 10b, c and also Fig. 11b, c we observe that BRPFC is more sensitive to the available cache than RPFC. This is because there is a single grammar in RPFC shared by all blocks while for BRPFC every block has its own grammar which results in reduced locality. Consequently, random lookups in BRPFC result in more cache misses than for RPFC. Sampled BRPFC performs similar to RPFC, because it also uses a single shared grammar for all blocks; hence, we do not report these numbers.

Their high cache locality results in very robust access times for *extract* and *locate* for both PFC and RPFC. Still, for BRPFC the response times degrade only by at most 2× if the cache size is reduced to 1/11-th of the cache. So even when a concurrent workload leads to high contention of the LLC, one can expect fairly robust access times for BRPFC. Considering the drastically reduced compression times of BRPFC and sampled BRPFC, the block-based compression schemes offer a good trade-off at slightly increased access times.

End-to-end experiments with real-world datasets

We also evaluate our compression method on three real-world datasets taken from SAP HANA scenarios which were also used in a previous study [35]. The first data set (ERP) is taken from an internal development system for the SAP ERP which is used to analyze mixed OLAP and OLTP workloads. The second data set (BW) is derived from customer data of a large warehouse system. The third data set (Mat) is a key column

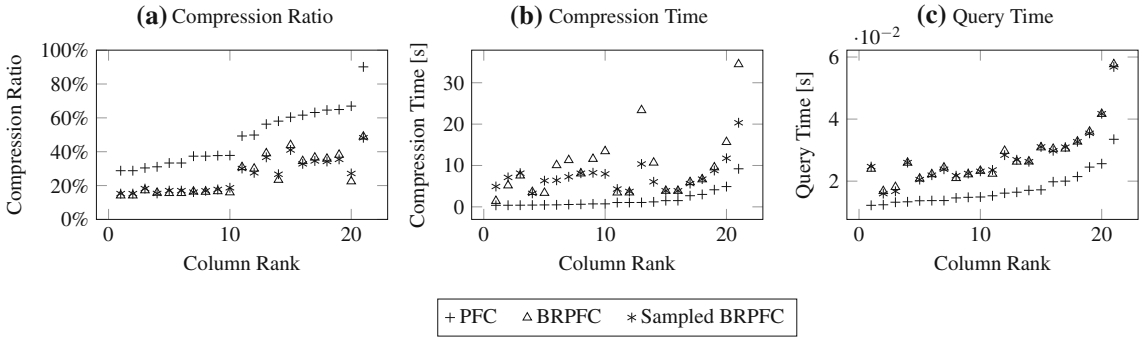

Fig. 12 Evaluation on real-world datasets

Table 7 Comparison of the evaluated compression methods

Compression method	Compression time	Compression quality	Vectorized *extract/locate*
RPFC [7]	Impractical for large datasets	1.95	
RPFC8	Impractical for large datasets	1.56	(✓)
RPFC8 16 bps	Impractical for large datasets	1.68	✓
BRPFC (ours)	Up to 30× faster than RPFC	1.70	✓
Sampled BRPFC (ours)	Up to 190× faster than RPFC	1.60	✓
PFC [7]	Fastest	1.00	n/a

of a very large table used in the SAP ERP for representing material management data including the current stock level for some material. As we are only interested in large columns containing strings, we ended up with 21 columns with an uncompressed size between 30.1 and 639.8 MB.

In Fig. 12, we present the results of our experiments with these real-world datasets. In each plot, we have ordered the measurements for a column by its rank using plain front coding. In Fig. 12a, we plot the compression ratio. These results are consistent with the other datasets as BRPFC and sampled BRPFC achieve a significantly better compression ratio than PFC. The largest dictionary in our dataset is compressed from 639.8 to 198.6 MB for PFC but to about 100 MB for BRPFC and sampled BRPFC. We show the *end-to-end time* for a delta merge in SAP HANA—which includes the time to compress the dictionary—in Fig. 12b. As BRPFC and sampled BRPFC add extra effort on top of PFC, their merge times are up to three times higher. However, this seems acceptable as the delta merge in SAP HANA is performed asynchronously and using a single background thread. In most cases, sampled BRPFC reduces the time for the delta merge compared to BRPFC—in the best case by 2.3× and for the column with the longest merge time from 34.5 to 20.3s. However, in the worst case, sampled BRPFC is 3.4× slower than BRPFC. This is due to the fixed overhead of compressing the superblock that sampled BRPFC has over BRPFC, which makes sampling not worthwhile to use for relatively small datasets. Finally, Fig. 12c presents the average *end-to-end time* to perform a query with a `LIKE` predicate that has to scan the full dictionary. Note, that multiple threads are used to scan for matches in the compressed dictionary in parallel. As expected, PFC's response times are lower because the other two methods add extra effort to decompress each string where BRPFC and sampled BRPFC have almost the same response times across all columns. In the worst case, the response times for BRPFC and its sampled variant are 2× higher than for PFC.

Discussion

As a conclusion to this evaluation, Table 7 shows a comparison of all evaluated string dictionary compression methods. The table compares compression time and quality for each method. The compression quality column shows the average compression rate achieved with each method, with PFC as the baseline, so higher values are better. It also lists whether vectorized access operations as described in Sect. 4 are supported. This is important, as the access times for Re-Pair based compression methods are not fast enough for general-purpose usage without vectorized decompression. For RPFC8, only the slower version of vectorized access operations is possible, because symbols are not fixed to a bitwidth of 16 here. Vectorized access operations as proposed in this paper do not apply to PFC, as it is not based on Re-Pair.

Of the methods presented here, we would generally recommend using sampled BRPFC with a superblock size of $S = 8$M, as it offers the best tradeoff between compression time and compression quality, while supporting fast vector-

ized access operations. However, if sampled BRPFC cannot provide the desired compression quality, BRPFC will yield better results at the cost of higher compression times than sampled BRPFC. For BRPFC, the block size can be varied to find an acceptable tradeoff between compression quality and time, like shown in Table 3.

Of course the cost of strong compression may not always be tolerable, for example for very frequently accessed columns, where faster response times outweigh the benefits of stronger compression. To automate the decision which string dictionary compression method shall be used for a given column, Müller et al. [8] propose the concept of a *compression manager*, which uses models of compression ratios, access and compression times for each available compression method to decide which method to use given the characteristics of the target column. This work could also be used to decide when to use one of the compression methods proposed here vs. other methods. The runtime model can be easily obtained using microbenchmarks such as the ones we performed in this evaluation. Due to the grammar-based nature of Re-Pair, it is hard to accurately predict compression ratios, however. The authors of [8] also noted this for the original RPFC method. They therefore proposed to model RPFC's compression ratios by assuming a uniform compression rate for the entire dictionary. This could also be done for our new methods.

6 Conclusion

In this paper, we have presented techniques that improve upon the access and compression times of the original RPFC string dictionary compression implementation by Navarro et al. [7]. To improve compression times, we proposed the block-based RPFC (BRPFC) method that applies Re-Pair independently on chunks of the input such that the working dataset during compression fits in the CPU caches. Additionally, we proposed a sampled variant of BRPFC, that resulted in compression times up to 190 times faster than RPFC. Furthermore, we showed that access times can be significantly improved by vectorizing the Re-Pair decompression routine with AVX-512. This makes *extract* and *locate* operations on average 2.3 and 1.8 times faster than those of the original RPFC, respectively. To facilitate vectorization and to achieve these speedups, we introduced two modifications to the compression algorithm. These cause decreased compression rates, but compression is still substantially better than that of plain front coding. Overall, this makes the proposed string dictionary compression methods in combination with the accelerated access operations attractive candidates for database workloads.

Acknowledgements Open Access funding provided by Projekt DEAL.

References

1. Abadi, D., Madden, S., Ferreira, M.: Integrating compression and execution in column-oriented database systems. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 671–682 (2006)
2. Lemke C., Sattler KU., Faerber F., Zeier A.: Speeding Up Queries in Column Stores. In: Bach Pedersen T., Mohania M.K., Tjoa A.M. (eds.) Data Warehousing and Knowledge Discovery. DaWaK 2010. Lecture Notes in Computer Science, vol 6263, pp 117–129. Springer, Berlin, Heidelberg (2010)
3. Westmann, T., Kossmann, D., Helmer, S., Moerkotte, G.: The implementation and performance of compressed databases. ACM Sigmod Rec. **29**(3), 55–67 (2000)
4. Färber, F., May, N., Lehner, W., Große, P., Müller, I., Rauhe, H., Dees, J.: The SAP HANA database—an architecture overview. IEEE Data Eng. Bull. **35**(1), 28–33 (2012)
5. Arz, J., Fischer, J.: LZ-compressed string dictionaries. In: 2014 Data Compression Conference, pp. 322–331 (2014)
6. Grossi, R., Ottaviano, G.: Fast compressed tries through path decompositions. J. Exp. Algorithm. (JEA) **19**, 3–4 (2015)
7. Martínez-Prieto, M.A., Brisaboa, N., Cánovas, R., Claude, F., Navarro, G.: Practical compressed string dictionaries. Inf. Syst. **56**, 73–108 (2016)
8. Müller, I., Ratsch, C., Färber, F.: Adaptive string dictionary compression in in-memory column-store database systems. In: EDBT, pp. 283–294 (2014)
9. Larsson, N.J., Moffat, A.: Off-line dictionary-based compression. Proc. IEEE **88**(11), 1722–1732 (2000)
10. Brisaboa, N.R., Cánovas, R., Claude, F., Martínez-Prieto, M.A., Navarro, G.: Compressed string dictionaries. In: International Symposium on Experimental Algorithms, pp. 136–147. Springer (2011)
11. Lasch, R., Oukid, I., Dementiev, R., May, N., Demirsoy, S.S., Sattler, K.U.: Fast & strong: the case of compressed string dictionaries on modern CPUs. In: Proceedings of the 15th International Workshop on Data Management on New Hardware, p. 4. ACM (2019)
12. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009)
13. Intel® Architecture Instruction Set Extensions and Future Features Programming Reference. https://software.intel.com/en-us/download/intel-architecture-instruction-set-extensions-and-future-features-programming-reference (2019). Accessed 09 Dec 2019
14. Intel® Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/ (2020). Accessed 08 May 2020
15. Willhalm, T., Oukid, I., Müller, I., Färber, F.: Vectorizing database column scans with complex predicates. In: International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures—ADMS 2013, pp. 1–12 (2013)

16. Willhalm, T., Popovici, N., Boshmaf, Y., Plattner, H., Zeier, A., Schaffner, J.: SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. Proc. VLDB Endow. **2**(1), 385–394 (2009)
17. Zhang, H., Liu, X., Andersen, D.G., Kaminsky, M., Keeton, K., Pavlo, A.: Order-preserving key compression for in-memory search trees. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (2020)
18. Clark, D.: Compact pat trees. PhD thesis, University of Waterloo (1998)
19. Kanda, S., Morita, K., Fuketa, M.: Practical string dictionary compression using string dictionary encoding. In: 2017 International Conference on Big Data Innovations and Applications (Innovate-Data), pp. 1–8. IEEE (2017)
20. Boncz, P., Neumann, T., Leis, V.: FSST: fast random access string compression. Preprint, available at https://github.com/cwida/fsst/raw/master/fsstcompression.pdf. Accessed 11 May 2020
21. Yasin, A.: A top-down method for performance analysis and counters architecture. In: 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 35–44 (2014)
22. Intel® VTune™ Amplifier. https://software.intel.com/en-us/vtune (2019). Accessed 09 Dec 2019
23. Gagie, T.I., Manzini, T., Navarro, G., Sakamoto, G., Takabatake, H.Y.: Rpair: rescaling repair with rsync. In: Brisaboa, N.R., Puglisi, S.J. (eds.) String Processing and Information Retrieval, pp. 35–44. Springer, Berlin (2019)
24. Kuipers, L., Niederreiter, H.: Uniform Distribution of Sequences. Courier Corporation, Chelmsford (2012)
25. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975). https://doi.org/10.1145/360825.360855
26. Intel® Xeon® Platinum 8180 Processor. https://ark.intel.com/content/www/us/en/ark/products/120496/intel-xeon-platinum-8180-processor-38-5m-cache-2-50-ghz.html (2017). Accessed 09 Dec 2019
27. Laboratory for Web Algorithmics—Datasets. http://law.di.unimi.it/datasets.php (2019). Accessed 09 Dec 2019
28. Boldi, P., Codenotti, B., Santini, M., Vigna, S.: Ubicrawler: a scalable fully distributed web crawler. Softw. Pract. Exp. **34**(8), 711–726 (2004)
29. Wikimedia Database Dumps. https://dumps.wikimedia.org/ (2019). Accessed 13 Jan 2020. Downloaded: 2019-03-06
30. GeoNames dump. http://download.geonames.org/export/dump/ (2019). Accessed 13 Jan 2020. Downloaded: 2019-03-06
31. Snappy, a fast compressor/decompressor. https://github.com/google/snappy. Accessed 05 April 2020
32. Snzip, a compression/decompression tool based on snappy. https://github.com/kubo/snzip. Accessed 05 April 2020
33. Yoshida, S., Kida, T.: A variable-length-to-fixed-length coding method using a re-pair algorithm. Inf. Media Technol. **8**(4), 971–977 (2013)
34. Intel Corporation: Improving Real-Time Performance by Utilizing Cache Allocation Technology. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf (2015). Accessed 13 Jan 2020
35. Moerkotte, G., DeHaan, D., May, N., Nica, A., Boehm, A.: Exploiting ordered dictionaries to efficiently construct histograms with q-error guarantees in SAP HANA. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 361–372. ACM (2014)