FSST: Fast Random Access String Compression

Peter Boncz
CWI
P.Boncz@cwi.nl

Thomas Neumann
TUM
neumann@in.tum.de

Viktor Leis
FSU Jena
viktor.leis@uni-jena.de

ABSTRACT

Strings are prevalent in real-world data sets. They often occupy a large fraction of the data and are slow to process. In this work, we present Fast Static Symbol Table (FSST), a *lightweight* compression scheme for strings. On text data, FSST offers decompression and compression speed similar to or better than existing speed-optimized methods such as LZ4, yet significantly better compression factors. Moreover, its use of a static symbol table allows *random access* to individual, compressed strings, enabling lazy decompression and query processing on compressed data. We believe these features will make FSST a valuable piece in the standard compression toolbox.

PVLDB Reference Format:
Peter Boncz, Thomas Neumann, and Viktor Leis. FSST: Fast Random Acc ess String Compression. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

In many real-world databases, including ERP [16] and visual analytics [19], a large fraction of the data is represented as strings. This is because strings are often used as a catch-all type for data of wide variety. In real-world databases, both human-generated text (e.g., description or comment fields) and machine-generated identifiers (e.g., URLs, email addresses, IP addresses, UUIDs, non-integer surrogate keys) are virtually always represented as strings.

Strings are often highly compressible. However, because strings often have many unique values, dictionary compression (which uniquely maps strings to fixed-size integers) is not always effective or applicable. Dictionary compression needs fully repeating strings to reduce size, and thus does not benefit when strings are similar but not equal. Also, dictionary compression is typically applied on limited chunks of data (a block, or row-group), in which case the dictionary must fit into a small part of the chunk; such that when there are many unique values, they do not fit the dictionary.

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

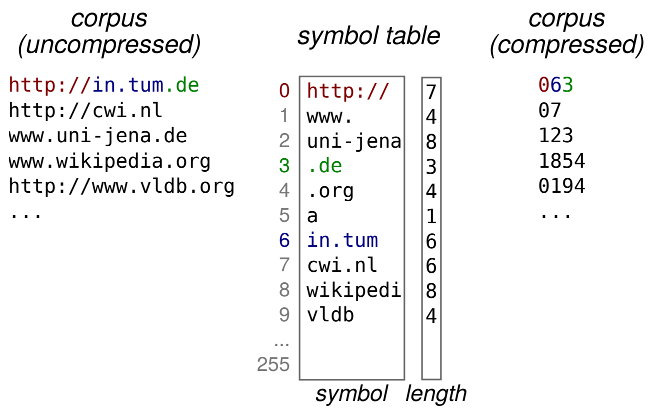


Figure 1: FSST provides very fast – yet effective – string compression, by replacing *symbols* of length 1-8 by 1-byte *codes*. Finding a good *symbol table* is a key challenge, addressed in Section 4. Techniques to make FSST compression very fast, including AVX512, are described in Section 5.

Most strings stored in databases are fairly small – generally less than 200 bytes and often less than 30 bytes per string. General-purpose compressors such as LZ4 and snappy are not suited for compressing small, individual strings, because they require input sizes on the order of several kilobytes for good compression factors. Some columnar database systems therefore use these general-purpose compression methods on coarse granularity, compressing columnar *blocks* of disk-resident data (i.e., compressing many string values together). However, database systems generally benefit from *random access* to individual string attributes, which is not possible when using block-wise general-purpose string compression. Examples of database access that requires random access to individual strings are: selection push-down in data scans (e.g., with strings stored in a columnar block), data access in joins and aggregations (e.g., with strings stored in a hash table) – or in fact any operator that does not consume *all* values in *sequential* order.

We present *Fast Static Symbol Table (FSST)* compression, a lightweight encoding scheme for strings. As is illustrated in Figure 1, the key idea behind FSST is to replace frequently-occurring substrings of up to 8 bytes with 1-byte codes. Decompression is very fast as it merely needs to translate each 1-byte code into a longer string using an array of 256 entries. These entries form an immutable symbol table that is shared among a block of string values, enabling decompressing individual strings. Previous random-access compression schemes were all very slow, while FSST is very fast in compression and decompression. The much lower speed of these random-access schemes may explain why they have not been widely adopted.

The key features of FSST are

- random access (the ability to decompress individual strings without having to decompress a larger block),
- fast decoding ($\approx$ 1-4 cycles/byte, or 1-4 GB/s per 3.6GHz core, depending on the data set),
- good compression factors ($\approx 2\times$) for textual string data sets, and
- high encoding performance ($\approx$ 4 cycles/byte, or $\approx$ 900 MB/s per 3.6GHz core).

In comparison with LZ4, which is so far the best general-purpose lightweight compression method (effective and much faster than e.g., snappy and zstd ([1]), FSST is better over all dimensions. FSST provides comparable – but often, faster – decompression and compression speed, and noticably better compression factors on textual data **on top of** its ability to compress and decompress strings individually, enabling random-access – which stands in contrast with all general-purpose methods (including LZ4) that only support efficient block-wise decompression. These features are useful in many applications, but are particularly useful in database systems. Fast compression and decompression enables all strings in the database to be stored in compressed form without significant performance loss, and being able to decompress individual strings enables fast point access (e.g., during an index lookup).

FSST can be integrated into existing data management systems and columnar file formats like Parquet, and should be used in conjunction with dictionary compression. In other words, after de-duplicating strings, FSST can be used to compress the unique strings within the dictionary. Given that strings make up a large fraction of real-world data [16, 19], this can have a substantial impact on overall space consumption. The source code of our highly-optimized C++ implementation is available under the MIT License:

`http://github.com/cwida/fsst`

The rest of the paper is organized as follows. In Section 2 we first describe related work on string compression, which is surprisingly sparse in comparison with the available research on integer compression. Section 3 then introduces the basic idea behind FSST and how decompression is implemented. The key algorithmic challenge of our approach is finding a good symbol table given a particular data set, for which we provide a genetic-like bottom-up algorithm in Section 4. FSST decompression is fast right out of the box, but making compression as fast as LZ4 is non-trivial. Techniques for this, including using AVX512 SIMD, are described in Section 5. Section 6 evaluates FSST using a wide variety of real-world string data showing that it offers good (de)compression speed and very good compression factors. Finally, we summarize the paper in Section 7.

2. RELATED WORK

Most research on lightweight compression for database systems concentrates on integer data [23, 9, 11, 18, 15, 6]. Similarly, work on query processing on compressed data generally does not focus on strings [20, 5]. We argue that given the prevalence and performance challenges of strings in real-world workloads [16, 10, 19], more research is required.

The most common approach for compressing strings is de-duplication using dictionaries [23, 9, 11, 18]. Dictionaries map each unique string to an integer code. The column then consists of these integer codes, which can additionally be compressed using an integer compression scheme. The strings themselves, which can make up the bulk of the data even after de-duplication, are not compressed in most database systems. In the following, we describe some of the proposals for compressing the string data itself.

Binnig et al. [4] propose an order-preserving string dictionary with delta-prefix compression. The dictionary is represented as a hybrid trie/B-tree data structure that stores the unique strings in sorted order. This order is exploited by the delta-prefix compression, which truncates the common prefixes of neighboring strings. To enable reasonably fast random access to individual strings, the full string is stored for every k (e.g., 16) strings. While delta-prefix compression is effective for some data sets (e.g., URLs), many other common string data sets (e.g., UUIDs) do not have long shared prefixes, which makes this scheme ineffective. Global dictionaries have additional downsides (e.g., more expensive updates) that have precluded their widespread adoption.

Another approach for compressing the string dictionary was proposed by Arz and Fischer [3], who developed a variant of LZ78 [22] that allows decompressing individual strings. However, with this approach decompression is fairly expensive, requiring more than 1 microsecond for strings with an average length of 19 [3]. This corresponds to roughly 100 CPU cycles per character or tens of megabytes per second, which is too slow for many data management use cases.

PostgreSQL does not use string dictionaries, but instead implements an approach called “The Oversized-Attribute Storage Technique” (*TOAST*). Values that are larger than 2 KB are compressed using a “fairly simple and very fast member of the LZ family of compression techniques” [2], and smaller values remain uncompressed. 2 KB is indeed a reasonable threshold for general-purpose compression algorithms, but short strings require a different approach.

Byte Pair [7, 21] is one of few compression schemes that allow decompressing individual, short strings. It first performs a full pass over the data, determining which byte values do not occur in the input and counting how often each pair of bytes occur. It then replaces the most common pair of bytes with an unused byte value. This process is repeated until there are no more unused bytes. In contrast to FSST's escaping scheme, Byte Pair's reliance on unused bytes implies that, in general, unseen data cannot be compressed given an existing compression table. The recursive nature of Byte Pair makes decompression iterative and – therefore – slow.

RePair [12] (Recursive Pairing) is a random-access compression format that recursively constructs a hierarchical symbol grammar. The initial grammar consists of all single-byte symbols, and is recursively extended by replacing the most frequent pair of consecutive symbols in the source text by a new symbol, reevaluating the frequencies of all of the symbol pairs with respect to the extended grammar, and

Algorithm 1 FSST-decoding

```
void decode(uint8_t*& in, uint8_t*& out,
            uint64_t sym[255], uint8_t len[255]) {
  uint8_t code = *in++;
  if (code != 255) {
    *((uint64_t*)out) = sym[code];
    out += len[code];
  } else { // escape code
    *out++ = *in++;
  }
}
```

Algorithm 2 FSST-encoding, given a symbol table.

```
void encode(uint8_t*& in, uint8_t*& out, SymbolTable& st) {
  uint16_t pos = st.findLongestSymbol(in);
  if (pos <= 255) { // no (real) symbol found
    *(out++) = 255;
    *(out++) = *(in++);
  } else {
    *(out++) = (uint8_t) pos;
    in += st.symbols[pos].len; // symbol length in bytes
  }
}
```

then repeating the process until there is no pair of adjacent symbols that occurs twice. Grammar construction in RePair is expensive, and the constructed grammar can be large and complex. Recent work improved RePair *decoding* speed using AVX512, but the reported throughput is still below 100MB/s [13], 20× slower than FSST; while encoding remains at least two orders of magnitude slower than FSST.

3. FAST STATIC SYMBOL TABLE

FSST's compression is based on the observation that, although each individual string might be short and have little redundancy, the strings of a column often have common substrings. To exploit this, FSST identifies frequently-occurring substrings, which we call *symbols*, and replaces them with short, fixed-size *codes*. Figure 1 illustrates this idea. For a URL corpus like the one shown in the figure, good symbols might be "http://", "www.", and ".org".

For efficiency reasons, symbols have a length between 1 and 8 bytes and are identified at byte (not bit) boundaries. Codes are always 1 byte long, which means there can be up to 256 symbols. However, one of the codes is reserved as an escape code as described in Section 3.2.

Given a particular data set, the compression algorithm first constructs a *symbol table* that maps codes to symbols (and vice versa). One crucial aspect of FSST is that the symbol table is the only state used during decompression and is immutable. This allows individual strings to be decompressed independently without having to decompress any other strings in the same compression block. General-purpose compression algorithms, in contrast, generally modify their internal state during compression and decompression, which precludes cheap point access.

3.1 Decompression

Given a symbol table and a compressed string, decompression is fairly simple. Each code is translated via an array lookup into its symbol and the symbols are appended to the output buffer. To make decompression efficient, we represent each symbol as an 8-byte (64-bit) word and store all symbols in an array. In addition, we have a second array that stores the length of each word. Using this representation, a code can be decompressed by unconditionally storing the 64-bit word into the output buffer, and then advancing the output buffer by the actual length of the symbol:

```
void decodeBasic(uint8_t*& in, uint8_t*& out,
                 uint64_t sym[256], uint8_t len[256]) {
  uint8_t code = *in++;
  *((uint64_t*)out) = sym[code]; // fast unaligned store
  out += len[code];
}
```

Relying on the fast unaligned stores that are available on modern processors, this implementation requires few instructions and is branch-free. It is also cache efficient as both the symbol table (2048 byte) and the length array (256 byte) easily fit into the level 1 CPU cache.

3.2 Escape Code

We reserve the code 255 as an escape marker indicating that the following byte in the input needs to be copied as is, i.e., without lookup in the symbol table. Note that having an escape code is not strictly necessary; it would also be possible to use only those bytes that do not occur in the input string as codes (as in the Byte Pair scheme discussed in Section 2). However, escaping has three advantages. First, it enables compressing arbitrary (unseen) text using an existing symbol table. Second, it allows symbol table construction to be performed on a sample of the data, thereby speeding up compression. Third, it frees up symbols that would otherwise be reserved for low-frequency bytes, thereby improving the compression factor.

Algorithm 1 shows the implementation of decoding with escaping. While this code contains a branch, it is well predictable since escape characters are rare in real-world data sets (otherwise the escaped input byte would have been included in the symbol table). Therefore, in practice, this version is faster than the version without escaping thanks to having a higher compression factor.

3.3 Compression

The algorithmic challenge of FSST is finding a symbol table for a given data set – we describe how to do this in Section 4. However, as Algorithm 2 shows, given a symbol table, the actual compression is conceptually straightforward. `findLongestSymbol` finds the longest matching symbol at the current input position. If no matching symbol was found, the input byte is escaped. Otherwise, the output is the code of the symbol found and the input position is incremented by the length of the symbol. Given the simplicity of the rest of the code, it is clear that the performance of compression is dominated by `findLongestSymbol`. Its implementation is described in Section 5.2.

3.4 Useful Properties

Strings stay Strings. Strings compressed in FSST become sequences of codes, i.e., sequences of bytes, so they effectively stay strings. This benefits the integration of FSST in existing (database) systems. Namely, already existing infrastructures to store strings can be re-used unchanged.

Compressed Execution. When querying a FSST-compressed database, one can *postpone* decompressing these values early in the query and do this only later. One reason that may force decompression is that some function or operator in the query actually needs to inspect the string values. Strings often face equality comparisons and a nice property of FSST is that such comparisons can be directly performed on the compressed value (even with the standard string equality function), as long as both operands are compressed with the same symbol table. Hence, in queries with an equality-selection predicate that compare a (compressed) table column against a constant, one can compress this constant and then process the predicate on compressed strings.

String Matching. It may be possible to perform more complex often-occurring string operations (e.g., `LIKE` pattern matching) on compressed strings as well, by the transformation of automata designed for their recognition in a byte-stream – re-mapping these onto a code-stream. This paper will not endeavor this route yet: we leave it to future work. However, it may not always be beneficial to perform costly operations on FSST-compressed strings, because FSST decompression is so fast; hence the compressed method should never become slower by more than the compression factor.

Late Decompression. If the operators that access the strings require their decompression, this can be done just before it is needed; all operators lower in the query plan can just store, copy and forward the compressed strings. The smaller size of the strings will make such manipulation faster, but it will also decrease the size of hash-tables, sort-buffers (reducing cache misses) and exchange spreading buffers (also reducing network traffic, in case of parallel and distributed query processing). Decoding strings on a remote computer may require sending the symbol table, which, as we will argue next, is small.

Small Symbol Table. Symbol tables have a maximum size of 8*255+255 bytes, but typically take just a few hundred bytes, because the average symbol length usually is around two. Thus, it is perfectly feasible to compress each page for each string column with a separate symbol table, but more coarse-grained granularities are also possible (per row-group, or the whole table). Finer-grained symbol table construction leads to better compression factors, since the symbol table will be more tuned to the compressed data. This does complicate the processing infrastructure for operating on compressed strings, since it needs to keep track which symbol table belongs to which string.

Parallelism. Since there is no (de)compression state, FSST (de)compression is trivial to parallelize – only the symbol table construction algorithm may need to be serialized. On the other hand, it may also be acceptable to have each thread that bulk-loads a chunk of data construct a separate symbol table (that should be put into each block header), such that compression also becomes trivially parallel.

0-terminated Strings. FSST optionally can generate 0-terminated strings (as used in C): code 0 then encodes the zero-byte symbol. Because in 0-terminated strings the zero-byte only occurs at the end of each string, there are effectively 254 codes left for compression. This slightly degrades compression (the 255-least valuable symbol has to be dropped from the symbol table, and its occurrences will be handled using escaped bytes), but this optional mode allows FSST to fit into many existing infrastructures.

4. SYMBOL TABLE CONSTRUCTION

The compression factor achieved by FSST on a data set depends on the 255 symbols chosen for the symbol table. We first discuss why constructing a good symbol table is challenging and then describe an effective bottom-up algorithm.

4.1 The Dependency Issue

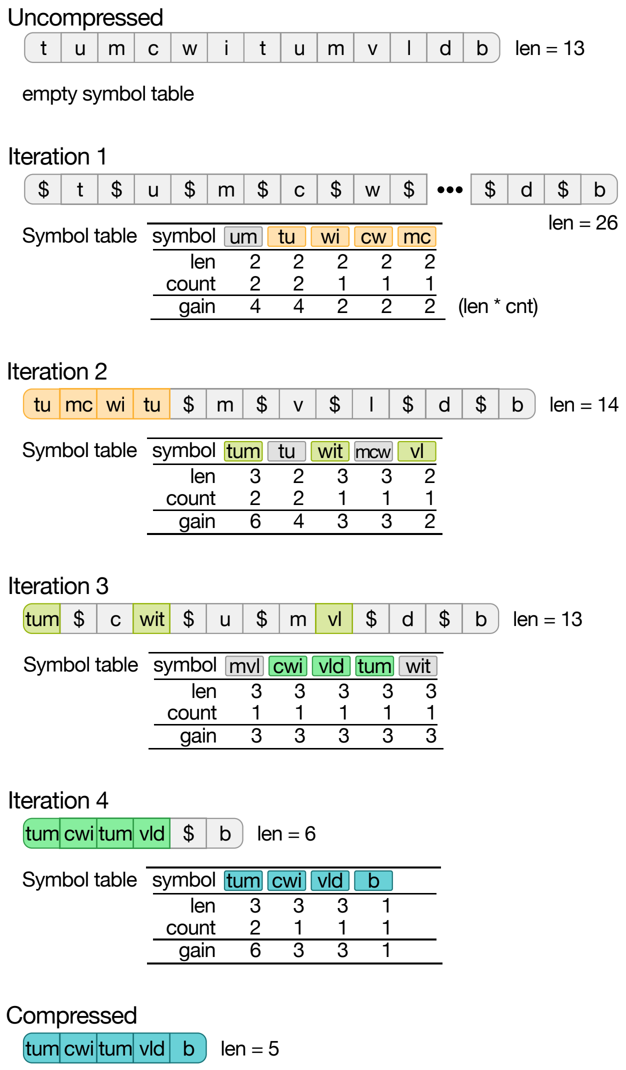
A naive, single-pass algorithm for constructing a symbol table would be to first count how often each substring of length 1 through 8 occurs in the data, and then pick the top 255 symbols ordered by *gain* (i.e., number of occurrences * symbol length). The problem with this approach is that the chosen symbols may *overlap*, and that the computed gains are therefore overestimates. In a URL data set, for example, the 8-byte symbol “http://w” might be chosen as the most promising symbol. However, the symbols “ttp://ww” and “tp://www” would seem equally promising, even though they do not improve compression once “http://w” has been added to the symbol table. Adding all three candidates to the symbol table would be a waste of the limited number of codes and would negatively affect the compression factor.

Another issue is that greedily picking the longest symbol during encoding does not necessarily maximize compression effectiveness. For example, if “http://w”, “<a href=”, and “"h” would be symbols in the symbol table, then the `encode()` method would **not** use the most valuable symbol “http://w” to encode the string “<a href="http://www.vldb.org”, because the symbol “"h” would have consumed the letter “h” already[1]. To summarize, symbol overlap combined with greedy encoding create the “dependency issue” between symbols that makes it hard to estimate gain and therefore to create good symbol tables.

Our first attempt at symbol table construction created a suffix array to identify the symbols with highest gain, but it produced significantly worse symbol tables than the algorithm we will present subsequently. Simple adaptions of our original solution that tried to correct for the overlap problem by subtracting gain from earlier chosen symbols, did not significantly improve the quality because they were overestimates. To reflect the dependency issue between symbols in compression, we call the gain computed based on frequency in the text *static gain*. The *actual* gain achieved by a symbol in compression is often significantly less.

To deal with the dependency issue, one could fall back to generating all possible symbol tables, testing them, and choosing the best one. The cost of testing one solution, is the cost of compressing the text, which is linear in its size N. However, finding an optimal solution (i.e., the 255 symbols that give the highest compression) is computationally expensive. The number of possible symbols to choose is bounded by $N * 8$ (i.e. a substring of length between 1 and 8, starting at any position) but that gives a bound on the number of symbol tables of $\binom{8N}{255}$. Being more practical, when creating a symbol table from a sample of a few tens of KB real-world text, one could narrow the problem down to choosing 255 from, for example, the top-3000 symbols in terms of static gain. The search space for symbol tables then still remains an unreasonable $\binom{3000}{255}$, a number with 378 digits.

[1] We experimented with an encoding function that frames string compression as a dynamic programming problem, rather than applying `findLongestSymbol()` greedily. However, we found that the compression factor only marginally improves, while encoding performance is severely affected.



symbol	um	tu	wi	cw	mc
len	2	2	2	2	2
count	2	2	1	1	1
gain	4	4	2	2	2

symbol	tum	tu	wit	mcw	vl
len	3	2	3	3	2
count	2	2	1	1	1
gain	6	4	3	3	2

symbol	mvl	cwi	vld	tum	wit
len	3	3	3	3	3
count	1	1	1	1	1
gain	3	3	3	3	3

symbol	tum	cwi	vld	b
len	3	3	3	1
count	2	1	1	1
gain	6	3	3	1

Figure 2: Four iterations of symbol table construction algorithm on the corpus "tumcwitumvldb" with a maximum symbol length of 3 and a maximum symbol number of 5. "$" stands for the escape byte.

4.2 Bottom-Up Symbol Table Construction

In the following, we present a bottom-up algorithm that has linear time complexity and overcomes the dependency issue using multiple iterations and on-the-fly compression.

The key idea behind our bottom-up algorithm is that the true worth of a collection of symbols is only learned *while compressing* the corpus with the symbols *interacting*. In other words, by counting the actually occurring codes in the compressed representation, we sidestep the dependency issue. Another building block is that, when evaluating a symbol table by compressing, one can also count the *pairs* of subsequent codes that manifest themselves to the compressor. The pair of two symbols is a candidate for becoming a new, longer symbol. Thereby, we refine the symbol table by concatenating frequently-occurring pairs of short symbols into new, longer, higher-gain symbols. To exploit these ideas, the symbol table construction algorithm performs multiple iterations in which it refines the symbol table. In each iteration, we add new promising symbols that then replace less worthwhile symbols from the previous iteration.

Our iterative algorithm starts with an empty symbol table. Each iteration consists of two steps: (1) we iterate over the corpus, encoding it on the fly using the current symbol table. This phase calculates the overall quality of the symbol table (the compression factor), but also counts how often each symbol occurs in the compressed representation, as well as each pair of successive symbols. (2) we use these counts to construct a new symbol table by selecting the symbols with the highest apparent gains.

We always consider all symbols from the previous generation, plus all new symbols generated by concatenating all occurring pairs of symbols. The new generation of symbols in the next iteration simply consists of the top-255 symbols considering their apparent gain in the previous iteration. Here, we mean with *considering*: computing the apparent gain (frequency*length) of a symbol, using the observed frequency the symbol being chosen *during actual compression*. In addition to pairs of symbols, we also (re-)consider all symbols that consist of a single byte, as well as consider extending each existing symbol with the next occurring byte – even if that single byte is not currently a symbol. The reason to always consider single byte(-extension)s is that it makes the algorithm more robust: creating a longer symbol from two shorter ones may cause the shorter symbol to disappear because the longer takes away some of its gain and thereby it may end up outside the top-255. In a next generation, this longer symbol may also lose the competition for survival; so that eventually a valuable symbol could disappear due to greedy combining. In essence, without reconsidering single byte(-extension)s, symbols would only grow longer, and going back to shorter symbols if that is better would never be possible. Continuously injecting single-byte symbol(-extension)s allows to "re-grow" valuable longer symbols that were lost due to such "too greedy" choices.

Figure 2 illustrates the algorithm by showing 4 iterations on the example corpus "tumcwitumvldb". To keep the example manageable, we assume that the maximum symbol length is 3 (rather than 8) and that the maximum symbol table size is 5 (rather than 255). After every iteration, we show the compressed string at the top, but instead of codes, for readability, we show the corresponding symbols. "$" stands for the escape byte. At the bottom of the figure, we show the symbol table, i.e., the top-5 symbols based on static gain. After the iteration 1, the top-5 symbols by static gain are "um", "tu", "wi", "cw", and "mc". In the first iteration the length of the compressed string temporarily doubles because the symbol table is initially empty and every symbol must be escaped. In iterations 2, 3, and 4, the quality of the symbol table steadily increases. After iteration 4, the corpus, which initially had a length of 13, is compressed to length of 5. The figure also shows that our algorithm makes mistakes, but that these are repaired in one of the next iterations. For example, in iteration 2, symbol "tu" looks quite attractive with a static gain of 4, but because "tum" is also in the symbol table, "tu" turns out to be worthless and is discarded in iteration 3.

4.3 Number of Iterations and Sampling

Our bottom-up approach means that we start with small symbols (size 1-2 after the first iteration), which grow over time (size 2-4 after the second and up to size 8 after the third iteration). Thus, *at least* three iterations are necessary to get to the maximum symbol length 8. Having larger symbols is of course crucial for good compression rations. We observed that 5 iterations are enough to converge to a good compression factor. For some data sets, more iterations (e.g., 10) slightly improve compression further, but this makes symbol table creation twice slower. Note that after each iteration we know how good the current symbol table is and we could stop early if convergence occurs.

In order to provide fast compression, symbol table construction should not take too much time. Because it encodes the data 5 times, and performs extra counting bookkeeping, symbol table construction should happen on a small sample of the data; if we assume it is 10× slower than encoding and want to limit its overhead to 10%, the sample should be smaller than a percent. Intuitively, using a sample instead of the full corpus works well because it is highly unlikely that symbols that occur frequently in the full corpus are uncommon in the sample. Experimentally, we indeed found that a fairly modest sample size results in compression factors close those of using the full corpus. Therefore, the compression utility we ship in our code uses a 16KB sample for compressing each 4MB chunk of string data.

5. OPTIMIZING COMPRESSION SPEED

The symbol table construction algorithm described in the previous section achieves good compression ratios, but, without additional optimizations, would be significantly slower than high-performance general-purpose compression implementations like LZ4. In this section, we describe our full highly-optimized compression implementation.

5.1 Symbol Table Construction

Algorithm 3 shows the code of the algorithm. The symbol table allows inserting a new symbol to the table (`add()`) and finding the longest matching symbol in a string (`findLongestSymbol()`). `buildSymbolTable()` is the main entry point of the algorithm and creates a new symbol table for a given string sample. It calls `compressCount()` (step 1 of the algorithm) and `makeTable()` (step 2 of the algorithm) to refine the symbol table five times.

`compressCount()` encodes the string sample on the fly while counting the occurrences of each code in the `count1` array and of each code pair in `count2` array. Exceptions are represented by 256 special symbols of length 1 (i.e., all bytes $0 \leq X < 256$) that have the negative code X-256. Normal symbols in the symbol table start with code 0 and grow upwards to a maximum <`nSymbols`≤255.

`makeTable()` iterates over `count1` and `count2`, inserting all symbol candidates into a priority queue. The frequencies `count1[]` collects for normal symbols, allow to calculate their *actual* gain, whereas the ones in `count2[]` speculate on having a symbol representing a concatenated pair of symbols in the symbol table – in reality this extra symbol would alter our greedy encoding decisions and would supplant another symbol, altering the experiment.

One small complication taken care of by `addOrInc()` is that when combining smaller symbols into longer ones, it is possible to generate the same longer symbol twice – both pairs (AB,C) and (A,BC) generate ABC – in which case the apparent gain is based on the sum of both `count2[]`s.

Our algorithm uses 5 iterations to converge to a good symbol table. Earlier versions used 10 iterations, but we reduced this to optimize symbol table construction time.

Algorithm 3 Bottom-up FSST Symbol Table Construction

```
struct Symbol {
  union val { char str[8], uint64_t num }; uint16_t len, code; };
struct SymbolTable {
 uint8_t nSymbols = 0; // number of normal symbols (<256)
 Symbol symbols[512]; // forall x: symbol[x].code = x-256
 //   0 <= x < 255: all single-byte symbols as escapes (negative codes)
 // 256 <= x < 256+nSymbols: normal symbols (the symbol table)
 // normal symbols are sorted by val.str (suffixing strings first)
 uint16_t sIndex[257] = {512}; // first normal symbol by startByte
 void add(Symbol s) {
  uint32_t pos = 256 + (s.code = nSymbols++);
  symbols[pos] = s;
  if (sIndex[s.val.str[0]] > pos) sIndex[s.val.str[0]] = pos;
 }
 uint16_t findLongestSymbol(char* buf){
  uint64_t nextWord = *(uint64_t*) buf, nextByte = nextWord&255;
  for (uint1t_t pos=sIndex[nextByte]; pos < sIndex[nextByte+1]; pos++)
   // uint64_t mask = (uint64_t) (-1LL>>8*(8-symbols[pos].len));
   // if (((symbols[pos].val.num ^ nextWord) & mask) == 0)
   if (!strncmp(symbols[pos].val.str, buf, symbols[pos].len))
    return pos; // first match is longest match
  return nextByte; // escaped byte
} };
SymbolTable buildSymbolTable(string sampl){ //MAIN METHOD
 SymbolTable st, bestTable = st; // create empty symbol table
 int gain, bestGain = 0;
 for (sampleFrac in {6, 30, 53, 77, 100} // use 5 passes on the sample
  uint16_t count2[512][512] = {0};
  gain = compressCount(st, count2, subSample(sampl,sampleFrac));
  if (gain > bestGain) { bestGain = gain; bestTable = st; }
  st = makeTable(st, count2);
 }
 return (gain > bestGain) ? st : bestTable;
}
// step 1: compute gain by counting code usage (+count code pairs)
int compressCount(SymbolTable st, uint16_t count2[512][512], string sampl){
 int gain = 0, *count1 = &count2[511][0]; // unused subarray
 uint8_t *cur = (uint8_t*) sampl.data(), *end = cur + sampl.size();
 for (uint16_t pos2,pos1=st.findLongestSymbol(cur); 1; pos1=pos2) {
  Symbol s = st.symbols[pos1];
  int isException = (pos1 < 256);
  gain += (s.len - 1) - isException; // gain = bytes saved
  count1[pos1]++; // count matched symbol
  if (s.len > 1) count1[*cur]++; // count single next byte also
  if ((cur += s.len)) >= end) break; // exit
  s = st.symbols[pos2 = st.findLongestSymbol(cur)];
  count2[pos1][pos2]++; // count symbol1 followed by symbol2
  if (s.len > 1) count2[pos1][*cur]++; // count symbol1,nextbyte also
 }
 return gain;
}
// step 2: create new symbol table from counts
SymbolTable makeTable(SymbolTable st, int count2[512][512]){
 int count1[] = &count2[511][0]; // used for single-symbol counts
 unordered_set<QSymbol> cands(); // candidate symbols & their value
 for (int pos1 = 0; pos1 < 256+st.nSymbols; pos1++) {
  Symbol s = st.symbols[pos1]; s.code = 0;
  addOrInc(cands,s,count1[pos]*s.len);
  if (s.len == 8) continue; // symbol cannot be extended
  for (int pos2 = 0; pos2 < 256+st.nSymbols; pos2++) {
   // concat: s.val.num |= st.symbols[pos2].val.num << 8*s.len;
   memcpy(s.val.str+s.len, st.symbols[pos2].val.str, 8-s.len);
   s.len = max(8, s.len + st.symbols[pos2].len);
   // clean: s.val.num &= (uint64_t) (-1LL >> 8*(8-s.len));
   addOrInc(cands, s, count2[pos1][pos2]*s.len);
 } }
 priority_queue<QSymbol,vector<QSymbol>,decltype<cmpGn>> pq(cmpGn);
 for (q : cands) pq.insert(q);
 set<Symbol,cmpStr> lt(); // sort selected symbols by string value
 for (;lt.size()<255 && !pq.empty(); pq.pop()) lt.insert(pq.top().sym);
 SymbolTable newTable; for (symbol : list) newTable.add(symbol);
 return newTable;
}
addOrInc(unordered_set<QSymbol> &cands, Symbol s, int gain){
 QSymbol q; q.gain = gain; q.sym = s;
 auto it = cands.find(q); // detect duplicate candidates (sum gain)
 if (it != cands.end()) { q.gain += (*it).gain; cands.erase(*it); }
 if (q.gain > 0) cands.insert(q);
}
struct QSymbol { Symbol sym; mutable uint32_t gain; }
bool cmpGn(QSymbol& l, QSymbol& r) { return l.gain < r.gain; }
bool cmpStr(Symbol& l, Symbol& r) { return l.val.num < r.val.num; }
```

As shown later in Figure 3, this reduces the compression achieved on our test corpus from 2.28 to 2.19 which is still quite good. Another effort-saving measure was *random subsampling* of the training data: we train the first iteration on a few percent, then linearly increase this sampleFrac in each iteration until it reaches 100% of our 16KB sample in the last iteration. Finally, we optimized the layout out the counter arrays, not only using short 16-bits or even 12-bits integers, but also splitting them in two arrays, one with the high bits and one with the low bits. This optimization (not shown in the listing) reduces CPU cache pressure, as only the low bits are updated frequently. As shown later in Figure 3, these measures reduced symbol table construction cost by a factor 2.2×, at the price of a slightly reduced compression factor. When creating a new symbol table for each 8MB block of string data, the amortized cost for symbol table construction as part of string compression is less than one CPU cycle per byte of input, which is generally less than 15% of compression cost.

5.2 Optimizing Encoding

Because constructing the symbol table involves repeatedly encoding the data, encoding performance influences not just the encoding phase itself but also symbol construction time. Our symbols have a maximum length of 8 bytes because this allows to represent them in a machine word; this can be seen in the `union` definition of the `Symbol` struct on top of the algorithm. A performance-critical method is `findLongestSymbol()`. We quickly narrow down the symbols to check to only those starting with a certain byte (using offset precomputed in `sIndex[]`), Note that we store the symbols ordered by string value, such that when one string is a prefix of another, it comes *after* the longer one. This particular ordering means that the first match is also the longest match, which is best for compression performance.

In Algorithm 3 we use the C-function `strncmp()` to check whether a prefix of the text matches a symbol. The two commented-out lines above show a much faster variant that performs the same check with an integer comparison – this is what we actually use in our code[2]. A further optimization that could be employed in `findLongestSymbol()` is for each symbol to materialize the `mask` value we compute in the optimized comparison. In order to fit the symbol table construction data structures into the CPU cache, the size of the `Symbol` record table should be kept small, so materializing a – redundant – 64-bits mask could not be worth it. We thus materialize the "ignoredBits" (the amount of bits to mask out, corresponding to unused characters in the 8-byte string): i.e., value (8*8-len) in the `Symbol`, a number between 0 and 63, which takes just 6 bits; to save computing two dependent operations (* and -) on our critical path.

Benchmarks on our test corpus show that encoding with `encode()`, that uses the `findLongestSymbol()` with integer optimizations, takes 20 cycles per input byte (on the i9-7900X CPU with g++ 8.3.1 -O3), which at a 3.6Ghz frequency translates into 200MB/s. While this is a quite usable speed already, it is clearly slower than LZ4, which achieves a 350MB/s compression speed for that task.

5.3 Encoding using Lossy Perfect Hashing

[2]Similarly, in `makeSymbolTable()` when two symbols get concatenated with `memcpy()`, in our code we use the commented out faster variant that uses integer operations.

The `encode()` performance mostly depends on the `findLongestSymbol()` method, which *iterates* over multiple symbols – all symbols that start with a certain byte – and checks for a prefix match. To accelerate encoding in general, and make it less vulnerable to symbol tables where many symbols share an equal first byte in particular, we modified FSST to get rid of this iteration. Rather than storing the symbols in a sorted array (indexed by `sIndex[]`), we switched over to a *perfect* hash table `hashTab[]`, plus a lookup array `shortCodes[]` (described later).

In a perfect hash table, there are no collisions and the hash computation immediately points to a bucket where the key should be, if present. Perfect hash tables normally need at least two hash functions and an additional offset array, which is used to eliminate hash collisions. To make encoding fast, we do not have time for computing two hash functions, and a memory access to an offset array; we just use a single multiplicative hash on the `symbol.val.num`. That is why we switch to a *lossy* approach: if two symbols are in a hash-collision with each other, we only keep the symbol with highest apparent gain. Rather than ending up with less than 255 symbols (due to throwing out collisions), we keep inserting symbols into the symbol table until it reaches 255 symbols. In other words, the penalty of throwing colliding symbols out, is alleviated by the bottom-up symbol generation mechanism that will find alternative, non-colliding symbols to fill the table.

The additional `shortCodes[X][Y]` array has 65536 entries (`X` and `Y` are bytes) used to check whether there is a 2-byte symbol `XY` that matches the next two bytes. If the array contains 511, its slot is free (escape); otherwise it contains the code of a normal symbol. However, in the high bits (bit 12-15) of these codes we also put the symbol length (1-8). Thus, casting the 16-bit code to a byte delivers the code (255 marks unused); the ninth bit tells whether it leads to an escaped byte, and (`code`>>12) contains the symbol length. After inserting the 2-byte codes in the array, we put in *all* free slots `shortCodes[Z][*]` the code of 1-byte symbols `Z`. Thus, we can use the array to check whether either a 2- or 1-byte symbol matches. The hash table key are the first three bytes of a symbol. Symbols with the same 3-byte prefix are therefore always collisions, in addition to hash collisions when two different 3-byte keys hash onto the same bucket.

The hash-based `encodeHash()` method is shown in Algorithm 4. As mentioned before, we materialize the amount of bits to mask out in comparisons, 8*(8-symbol.len), as "ignoredBits" in the `Symbol` record. Rather than using separate byte fields, we store ignoredBits, code and length (I, C, L) in a field called `icl`; mostly to entice the compiler to access all of these values with a single load instruction. As such, casting `s.icl` to a byte produces `s.ignoredBits`, similarly casting `icl`>>16 produces `s.code`. Note that this code representation in the hash table is bitwise compatible with the one in `shortCodes`: it also contains an escape marker in the ninth bit and the symbol length in the high bits.

Algorithm 4 (lowest three lines) reads the next 8 bytes of the to-be-encoded string into `word`, and looks up the short code for it in `code`, looking only at the first two bytes. Then `hashKernel()` first calls `hashFind()` that uses the next three bytes of input to probe the hash table. However, as a side-effect we there also store the first input byte into the second output position already. This eventually would only need to be done for escaped bytes, but we do it always, because

Algorithm 4 encoding with Lossy Perfect Hashing

```
struct Symbol {
   union val { char buf[8]; uint64_t num};
   uint32_t icl; // [0..8]=ignoredBits, [16..25]=code, [26..32]=len
}
struct SymbolTable {
  uint8_t nSymbols; // # of normal symbols (not counting escapes)
  Symbol symbols[512]; // all symbols: 0-255 escapes, then n Symbols

  uint16_t shortCodes[256][256]; // codes for 1- and 2-byte symbols

  Symbol hashTab[hashTabSize]; // keyed on the first three bytes
  static uint64_t hashTabSize = 4096; // fits L1

  // symbol reordering for fast encoding: 2-byte first, 1-byte last
  // 2-byte symbols that do not have a longer symbol as suffix..
  uint8_t suffixLim; // ..have code < suffixLim
  uint8_t byteLim; // codes > this are for 1-byte symbols
  // very fast and simple multiplicative hash
  uint64_t hash(uint64_t x) { return (x*2971215073)^(x>>15); }
}

void
encodeHash(uint8_t*& cur, uint8_t*& out, SymbolTable& st){
  Symbol s;
  auto hashFind = [&]() {
    out[1] = word; // dirty trick: speculatively write out escaped byte
    uint64_t idx = hash(word & 0xFFFFFF) & (st.hashTabSize-1);
    s = st.hashTab[idx]; // fetch symbol from hash table
    uint64_t clean = (-1LL >> /*ignoredBits*/ (uint8_t) s.icl);
    return (s.icl < 0xFFFFFFFF && s.val.num == (word & clean));
  };
  auto hashedKernel = [&]() {
    if (hashFind()) {
      cur += s.icl >> 28; // len
      *out++ = (uint8_t) (s.icl >> 16); // code byte
    } else if ((uint8_t) code >= st.byteLim) {
      cur++; // 1-byte code or escape.
      *out = (uint8_t) code;
      out += 1+((code>>8)&1); // predicated increment by +1 or +2
    } else {
      cur += 2; // 2 byte code
      *out++ = (uint8_t) code;
    }
  );
  uint64_t word = *(uint64_t*)cur;
  uint64_t code = st.shortCodes[word & 0xFFFF];
  hashedKernel();
```

this write does not hurt and can be done without any dependencies. `hashFind()` looks up the stored symbol (`s`) in the hash table, and cleans `word` by zero-ing out the irrelevant bytes w.r.t. the symbol we found (for symbols shorter than 8). It tests if the hash table slot actually is non-empty (`s.icl<0xFF..`) and whether the `word` matches the symbol `s`. If all of this returns a hit, `hashKernel()` emits the code coming from the hash table as output and advances the output pointer by the length of the symbol. Otherwise, it checks if the short code was a 1-byte symbol. This check is done by comparing if the `code`$\geq$`byteLim`. Namely, in the symbol table we now rearrange the codes, putting all the 1-byte codes (the very shortest ones) last. This is purely done to make this test simple. Because, a 1-byte advance in the input is either made by a 1-byte symbol, or due to an escaped character. Therefore. the 1-byte codes must be the highest ones (close to 255) to test both cases in a single comparison. Every instruction counts on the hot-path of encoding.

Note that in case of an escape code, the output pointer needs to be incremented by two positions, not one. This decision is made without a branch, by adding the ninth bit of the `code` field to the increment. All other cases (not in perfect hash table, not 1-byte) are 2-byte codes, which are handled in the obvious way.

Algorithm 5 Kernel for `encodeSingleBranch()`

```
auto singlebranchKernel = [&](){
  if (hashFind()) {
    cur += s.icl >> 28; // advance by symbol length
    *out++ = (uint8_t) (s.icl >> 16); // code byte
  } else {
    // handle 1- and 2-byte symbols with predication
    cur += (code>>12); // advance by symbol length
    *out = (uint8_t) code; // code or 255=escape
    out += 1+((code>>8)&1); // predicated +1 or +2
  }
};
```

Algorithm 6 Short Symbol Optimization

```
void encodeShort(uint8_t*& cur, uint8_t*& out, SymbolTable& st){
  uint64_t word = *(uint64_t*)cur;
  uint64_t code = st.shortCodes[word & 0xFFFF];
  if (uint8_t) code) < st.suffixLim) {
    cur += 2; // 2-byte code without having to look at hash table
    *out++ = (uint8_t) code;
  } else hashedKernel();
}
```

Benchmarks on our test corpus show that `encodeHash()`, takes 12.4 cycles per input byte (on the i9-7900X CPU with g++ 8.3.1 -O3), which at a 3.6Ghz frequency gives 290MB/s.

5.3.1 *Adaptive Encoding Kernel*

We further created two alternative ways of encoding efficiently: `encodeSingleBranch()` and `encodeShort()`, shown in resp. Algorithm 5 and Algorithm 6. The single-branch variant allows to handle all 1- and 2-byte symbols without any branch; there is only an if to decide between looking in the `hashTab[]` and the `shortCodes[]` array. To allow so, we retrieve the high code bits `code>>9`, which contains symbol length and increment the input pointer by that.

On the other hand, the `encodeShort()` method works best when most codes are 2-bytes long (thus do not stem from the hash table). Such symbol tables are found in text with high entropy, like hexadecimal encrypted passwords. This kernel quickly tests whether the short-code is a 2-byte symbol for which there exists no other longer symbol that is a suffix. In that case, we can skip testing the hash table altogether. To allow for this comparison, we give all symbols that have this property the lowest codes.

The question is now: which encoding variant is best? We could use *micro-adaptivity*[17] to decide this at run time based on performance measurements. The FSST library now uses a simpler solution that makes the decision based on symbol table properties, optimizing CPU branch prediction efficiency. If 65 percent of the symbols are 2-bytes long and 95 percent of those do not have a longer extension, we use `encodeShort()`. If the amount of 1-byte codes is low (between 5 and 20 percent) we use `encodeSingleBranch()`; in all other cases we use `encodeHash()`.

This adaptive algorithm reduces encoding cost to 10.2 cycles per tuple, which equates on our machine to 350MB/s. It almost doubles the 200MB/s of the original algorithm, and the perfect hashing approach comes at a loss of only 5% in the average compression factor.

The lossy perfect hash approach modifies the FSST format in that the codes in the symbol table must be organized such that all 1-byte symbols have the highest codes. On the other hand, the 2-byte codes for which there is no symbol that is a suffix, must have the lowest codes. Also, it requires the symbols longer than 3-bytes to be non-colliding in the hash

Algorithm 7 Kernel for `encodePredicated()`

```
auto predicatedKernel = [&](){
  code = hashFind() ? (s.icl>>16) : code; // conditional move
  *out = code; // write out code byte (or 255)
  out += 1+((code>>8)&1); // increase with 1 or 2 (escape = 9th bit)
  cur += (code>>12); // symbol length is in bits [12..15] of code
};
```

table. This property is needed in order to support adding compressed data to a data set with an existing symbol table.

5.4 AVX512 SIMD Implementation

In Algorithm 7 we show a scalar FSST-compression kernel using *predication* (turning all if-then-else's into computations). It exploits the fact that `shortCodes[]` and the perfect hash table store information in the same format, i.e. `code` and `icl>>16` have the code in the low byte, the ninth bit indicating an escaped character and the symbol length in the high bits. Choosing between the two is done using an `X=cond?Y:Z` construct, which compilers often handle with a branch, but which also can be handled with a *conditional move* instruction – also supported in AVX512. The scalar `predicatedKernel()` is not fast at all, but it provides the model for the AVX512 version of FSST-encoding.

An important idea is to compress *multiple string segments* in parallel in 8 different 64-bits AVX512 lanes. The maximum segment size is 511-bytes: longer strings are chopped up by enclosing scalar code (not shown). We unroll the kernel 3x (not shown), so 24 strings are compressed at-a-time.

An issue that we glossed over in the algorithms given so far is dealing with end-of-string correctly. The hash-based kernels all can jump over end-of-string, as they blindly load the next 8 input bytes. Scalar code could deal with this by testing whether `cur<end-7`, and use a slower variant to encode the last 1-7 bytes. However, in SIMD we must avoid all branches. FSST deals with this by introducing a *terminator* byte. This is a byte that cannot be part of a symbol longer than 1. Thus, if a terminator byte is put at the end of the to-be-encoded string, matching cannot jump over it.

We use the byte with the lowest frequency in the input corpus as terminator – except when in 0-terminated mode, because then byte 0 is the terminator. The terminator character is appended to each 511-byte segment in the enclosing scalar code that calls the AVX512 kernel (not shown).

The `fsst_compress()` API function compresses a batch of strings, preferably 100 or more (it can also be fewer, or even just one string, if the total string volume is significant – a few tens of KB or more). The strings are copied into a temporary buffer of 512 segments, chopping up long strings if needed. When 512 segments have been gathered or there is no more data, the AVX512 encoding kernel in Algorithm 8 is invoked. Each segment is an *encoding job*. The while-loop in Algorithm 8 encloses the AVX512 encoding kernel: each iteration it finds in each of the strings (=lanes) the next symbol, advancing 1-8 bytes in the input, and 1 byte in the output (or 2, in case of an escape). This loop finishes when there are no longer enough active jobs to fill the lanes (here:8, but in the 3x unrolled implementation 24). When we run out of jobs and revert to scalar code, we also finish the still active jobs in scalar mode.

Algorithm 8 AVX512 FSST-encoding kernel (not unrolled)

```
int encodeAVX512(
 SymbolTable &st, int njobs,
 uint64* injobs, *outjobs, // arrays with max 512 jobs
 char *input, *output) // tempstring buffers (resp 256KB,512KB)
{
  char *hashTab=(char*)st.hashTab, *shortCodes=(char*)st.shortCodes;
  uint64* lastjob = injobs+njobs; // points to end of injobs
  __mm512_i write, job;//job bit-format: [out:19][nr:9][end:18][cur:18]
  __mm512_i cur, end, len, word, code, esc, idx, hash, icl, symb;
  __mmask8 match, loadmask=255;
  int done=0, delta=8;

  while(injobs+delta < lastjob) { // while all lanes busy
    // fetch 8 jobs. in this kernel we will find 1 code for each
    job = _mm512_mask_expandloadu_epi64(job, loadmask, injobs);
    injobs += delta;

    // current position in each string
    cur = __mm512_srli_epi64(job, 19+9+18); // cur field at bit 46

    // get 8 bytes from the input strings
    word = _mm512_i64gather_epi64(cur,input,1);//1=byte addressing

    // code = shortCodes[X][Y]
    // constants x8_YY: hexidecimal value YY in all 8 (64-bits) lanes
    idx = _mm512_and_epi64(word, x8_FFFF);
    code = _mm512_i64gather_epi64(idx, shortCodes, 2);

    // speculatively put first byte into second position of write reg
    write = _mm512_slli_epi64(_mm512_and_epi64(word,x8_FF), 8);

    // idx = first three bytes of string, hash fetch into icl
    idx = _mm512_and_epi64(word, x8_FFFFFF);
    hash = _mm512_mullo_epi64(idx,x8_PRIME);//YY=2971215073
    idx = _mm512_xor_epi64(hash, mm512_srli_epi64(idx, 15));
    idx = _mm512_and_epi64(idx, x8_MASK); // MASK=4095
    idx = _mm512_slli_epi64(idx,4); // multiply idx*16 (bucket width)
    icl = _mm512_i64gather_epi64(idx,hashTab,1);//probe hash table

    // fetch the symbol (text) part of the hash table record and compare
    symb = _mm512_i64gather_epi64(idx,hashTab+8,1);//next 8bytes
    match = _mm512_cmplt_epi64_mask(icl, x8_FF0000); // used?
    icl = _mm512_and_epi64(icl,x8_FF); // get ignoredBits
    icl = _mm512_srlv_epi64(x8_FFFFFFFF, icl); // use it to mask
    word = _mm512_and_epi64(word, icl); // clean the word
    match &= _mm512_cmpeq_epi64_mask(symb, word); // match?
    icl = _mm512_srli_epi64(icl, 16); // extract code+len from icl

    // select between code+length from shortCodes and hashTab (match)
    code = _mm512_mask_mov_epi64(code, match, icl);

    // put code byte into write register, and scatter write to output
    write=_mm512_or_epi64(write,_mm512_and_epi64(code,x8_FF));
    idx=_mm512_and_epi64(job,x8_7FFFF);//get 19-bit output offset
    _mm512_i64scatter_epi64(output, idx, write, 1);

    // job bookkeeping: advance cur and out
    code = _mm512_and_epi64(code, x8_FFFF);
    len = _mm512_srli_epi64(code, 12); // get symbol length from code
    job = _mm512_add_epi64(job, _mm512_slli_epi64(len, 46));
    esc = _mm512_srli_epi64(code, 8); // shift away 8 bits
    esc = _mm512_and_epi64(esc, x8_1)); // keep only 9th bit
    job = _mm512_add_epi64(job, _mm512_add_epi64(x8_1, esc));
    cur = _mm512_srli_epi64(job, 19+9+8); // cur field at bit 46
    end = _mm512_srli_epi64(job, 19+9); // end field at bit 28
    end = _mm512_and_epi64(end, x8_3FFFF); // keep 18 bits

    // write out ready jobs
    loadmask = _mm512_cmpeq_epi64_mask(cur, end);
    _mm512_mask_compressstoreu_epi64(outjobs+done,loadmask,job);
    done += (delta = _mm_popcnt_u32((int) loadmask));
  }
  // flush active and unprocessed jobs
  __mmask8 activemask = 255 & ~loadmask;
  _mm512_mask_compressstoreu_epi64(outjobs, activemask, job);
  int i=done+8-delta;
  while (injobs < lastjob)
    outjobs[i++] = *injobs++;
  return done; // outjobs[done..njobs-1]: 2b finished with scalar encoding
}
```

A job-description is a 64-bits integer consisting of two input string offsets (`cur` and `end`). Their widths are 18 bits, so their maximum value is 256K, which is the size of the segment buffer (512x512). The output offset `out` is 19-bits, since in worst case FSST produces output that is twice the input. Because some jobs will stay longer in the processing kernel than others, they will not finish in the input order and it is necessary to track the job number `nr`: a 9-bit number (we have max. 512 jobs). Note that 18+18+19+9=64.

The reason to squeeze all this control information into a single lane is AVX512 register pressure. By carefully using as few registers as possible, it is possible to unroll this kernel 3x without suffering performance degradation due to register spilling. Unrolling AVX512 gather and scatter instructions is necessary, because they have a very long latency (upwards of 25 cycles) yet multiple executions can be overlapped (three). Note that even used in overlapped AVX512 mode, gather instructions just load (from the CPU cache) around 1 word per cycle amortized; whereas modern Intel processors can load two words per cycle with scalar loads. The strength of AVX512 is not memory access, but parallel computation, which we leverage in this compression kernel.

We do not just fire off the SIMD kernel once to process 8 strings in its 8 lanes (or 24 strings in 24 lanes, 3x unrolled), because some strings will be much shorter than others and some will compress much more than others. This would mean that many lanes would be empty towards the end of encoding work. Therefore we buffer 512 jobs and refill the lanes in each iteration, when needed, from that job buffer.

We first radix-sort the job control array on reverse string length – quickly, in a single pass – so the longest strings start being processed first, helping load balancing. Jobs may finish in a non-sequential ordering anyway, so starting encoding work in non-sequential job order due to sorting does not complicate the algorithm (any further).

The AVX512 encoding kernel on our benchmark runs each iteration in about 200 cycles, and given that 24 strings are processed in parallel, and each step we advance roughly 2 bytes in each, this translates to about 4.1 cycles per byte, which on our i9-7900X equates to 910MB/s. As such, FSST is the fastest known string compressor available.

This AVX512 encoding kernel again led to slight adaptations of the FSST format. Not only the SIMD, but also the scalar default `encode()` method use the 511 byte input segmenting. We use the scalar variant on architectures that do not support AVX512, but also to encode short strings (shorter than 10 bytes). Using input string segmenting in scalar code is necessary to preserve the property that two strings encoded with the same symbol table are binary identical, such that string equality can be executed on compressed strings. Further, the scalar `encode()` also uses the terminator byte as a fast way to avoid going over end-of-string. The terminator byte is meta-data that is added to the symbol table, such then when we serialize and deserialize a symbol table (for persistent storage, or distributed processing), this information is preserved.

A final question could be whether SIMD could also be useful for decompression. We think it is not. Given the already very high decompression speed of `decode()` and its characteristics of having very little computational effort and consisting only of memory instructions (see Algorithm 1), it does not make sense to pursue SIMD for decompression.

Table 1: Evaluation data sets ("dbtext" corpus) with their LZ4 and FSST compression factors.

	avg		LZ4	FSST
name	len	example string	factor	factor
hex	8	DD5AF484	1.14×	2.11×
yago	19	Ralph_A._Brown	1.25×	1.63×
email	22	xnj_14@hotmail.com	1.55×	2.13×
wiki	23	Benzil	1.31×	1.63×
uuid	37	84e22ac0-2da5-11e8-9d15- . . .	1.55×	2.44×
urls2	55	http://fr.wikipedia.org/ . . .	1.75×	2.05×
urls	63	http://reference.data.go . . .	2.77×	2.42×
firstname	7	RUSSEL	1.25×	2.04×
lastname	10	BALONIER	1.28×	1.97×
city	10	ROELAND PARK	1.37×	2.14×
credentials	11	PHD, HSPP	1.48×	2.31×
street	13	PURITAN AVENUE	1.60×	2.35×
movies	21	Return to 'Giant'	1.23×	1.66×
faust	24	Erleuchte mein bedÃijrftig Herz.	1.48×	1.87×
hamlet	30	<LINE>That to Laertes . . .	2.13×	2.41×
chinese	87	道人决心消除肉会 . . .	1.40×	1.69×
japanese	90	せん。しかし、. . .	1.84×	2.00×
wikipedia	130	Weniger hÃďufig fressen sie . . .	1.45×	1.81×
genome	10	atagtgaag	1.59×	3.32×
location	40	(40.84242764486843, -73 . . .	1.58×	2.51×
c_name	19	Customer#000010485	3.08×	3.80×
l_comment	27	nal braids nag carefully expres	2.22×	2.90×
ps_comment	124	c foxes. fluffily ironic . . .	2.79×	3.40×

6. EVALUATION

FSST has been designed for compressing textual string columns. To evaluate it, we collected a text corpus (dubbed "dbtext") consisting of 23 string columns covering a wide variety of real-world string data. The columns used in the evaluation are shown in Table 1 and can be categorized to into

- machine-readable identifiers (hex, yago, email, wiki, uuid, urls2, urls),
- human-readable names (firstname, lastname, city, credentials, street, movies),
- text (faust, hamlet, chinese, japanese, wikipedia),
- domain-specific codes (genome, location), and
- TPC-H data (c_name, l_comment, ps_comment).

The average string length per column ranges from 7 (firstname) to 130 (wikipedia). Most of the data comes from real-world sources like Wikipedia, Tableau Public [8], or IMDB [14]. For reproducibility, the data sets will be published together with our C++ source code. Note that some of these columns (e.g., hex, uuid, genome, location) would be better represented using specialized data types. However, industrial experience taught us that users virtually always use the string data type in these cases [19].

All experiments were performed on a workstation with 32GB RAM and a single 10-core 3.3GHz i9-7900X CPU, which has two AVX512 execution units per core. This server is running Linux (Fedora Core) 4.18.16. We used the latest version of LZ4 (1.9.2), and compiled all code with g++ (8.3.1) and flags -O3 -march=native. We use single-threaded execution, but note that both compression and decompression can trivially be parallelized by splitting the data into independent blocks (row-groups).

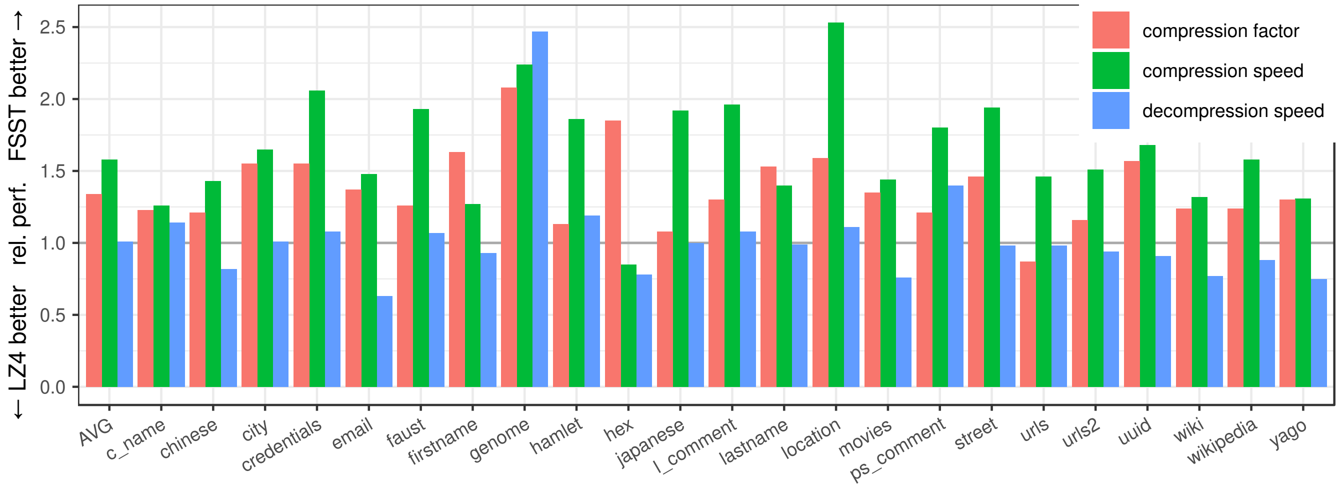


Figure 3: Relative performance of FSST versus LZ4 in terms of compression factor, compression speed, and decompression speed. Each data set is treated as a 8MB file.

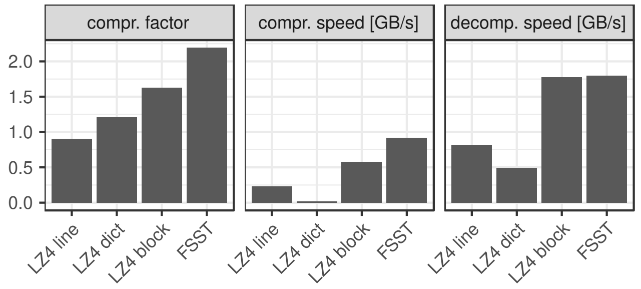


Figure 4: With LZ4 short strings do not compress well, even with a pre-generated dictionary.

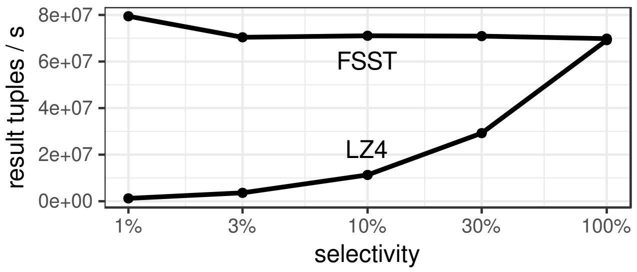


Figure 5: Selective queries are fast in FSST due to random access to individual values.

6.1 File Mode

Let us first compare FSST with LZ4, which is currently the best general-purpose lightweight compression implementation. In this experiment we treat each string column as a file, concatenating all strings until each file has 8MB of data. Note that this file-based mode is the best case for LZ4, since it has large blocks to compress and we not exploit FSST's random access capability. We measure the compression factor, bulk compression speed, and bulk decompression speed.

As Table 1 shows, the compression factors for FSST range from 1.63× (wiki) to 3.80× (c_name), with an average of 2.19×. Thus, as a rule of thumb, FSST halves space utilization for textual data. LZ4, for comparison, achieves an average compression factor of 1.63×.

Figure 3 shows the relative performance of LZ4 and FSST on the three metrics for each data set individually and on average. For almost all data sets, FSST outperforms LZ4 in terms of the compression factor and compression speed. On average, besides resulting in a 34% higher compression factor, FSST also achieves 58% higher compression speed. For decompression speed, FSST is faster on some data sets and LZ4 is on others – with the average being almost identical (FSST is faster by 1%).

6.2 Random Access

In database scenarios we typically do not store large files but instead we have string attributes or dictionaries with a large number of relatively short strings. Compressing these strings individually with LZ4 gives a very poor compression factor, as shown in Figure 4. Plain LZ4 (*LZ4 line*) cannot handle the short strings reasonably – the compression factor is below 1, meaning that the data size actually slightly *increases*. LZ4 also optionally supports using an additional dictionary, which needs to shipped with the compressed data. Using zstd to pre-generate a suitable dictionary for the corpus (*LZ4 dict*) improves the compression factor a bit, but hurts the compression speed very severely. The only meaningful way to use LZ4 for string attributes is to compress blocks of 1,000 values together (*LZ block*), which helps compression but prevents random access. FSST offers much better compression factors and compression speed than all LZ4 variants, and decompresses just as fast as the fastest LZ4 variant.

Note that the block mode of LZ4 is not ideal for database applications. When selecting only a subset of the values, one still has to decompress the whole block for LZ4, while FSST offers random access. The effect of this is shown in Figure 5. When retrieving a subset of values from a compressed relation, the output rate of FSST is unaffected by the selectivity, while LZ4 block has to decompress all values, including values that are not needed for the result. This makes FSST much more attractive for database use cases.

Table 2: Detailed FSST encoding performance as cycles-per-input-byte for various encoding kernels. Left: "simd$_3$" is fastest, i.e., AVX512 using 3-way enrolling of the kernel in Algorithm 8. Right: the adaptive kernel ("adaptv") automatically chooses the scalar kernel with minimum cost ("short" is the ShortSymbol kernel of Algorithm 6, "single" the single-branch kernel of Algorithm 5, and "hash" the original lossy perfect hashing kernel that uses if-then-elses of Algorithm 4).

$simd_1$	$simd_2$	$simd_3$	$simd_4$	*short*	*single*	*hash*	*adptv*	
8.01	5.36	**4.98**	5.16	16.13	11.61	12.39	**11.26**	firstname
7.97	5.08	**4.17**	4.42	4.90	8.75	6.24	**5.09**	hex
8.70	5.51	**5.07**	4.69	15.09	11.04	11.63	**10.82**	city
5.29	3.57	**2.85**	3.26	8.96	8.89	8.97	**9.00**	genome
8.05	5.12	**4.18**	4.44	15.15	10.50	11.03	**10.35**	lastname
7.82	5.03	**4.68**	4.61	15.75	13.70	14.60	**13.41**	credentials
7.35	5.02	**4.36**	4.52	14.71	11.56	12.90	**11.46**	street
3.87	2.76	**2.23**	2.42	6.60	6.22	5.53	**5.37**	c_name
9.36	5.91	**5.03**	5.31	14.68	11.56	13.56	**11.50**	yago
9.10	5.74	**4.82**	5.14	14.34	11.49	13.27	**11.36**	movies
7.19	4.60	**4.03**	4.25	12.09	10.11	11.54	**10.13**	email
9.37	5.92	**4.93**	5.30	14.07	11.88	13.51	**11.88**	wiki
8.55	5.51	**4.98**	5.18	15.10	13.42	14.46	**13.05**	faust
5.59	3.63	**3.16**	3.42	11.09	9.78	10.73	**9.78**	l_comment
7.35	4.56	**4.13**	4.23	11.84	10.36	11.22	**10.50**	hamlet
6.36	4.08	**3.40**	3.59	9.64	8.35	8.04	**8.06**	uuid
5.64	3.70	**3.05**	3.14	9.99	8.02	5.74	**5.73**	location
7.50	4.72	**4.03**	4.28	10.33	10.52	10.14	**10.12**	urls2
6.32	4.04	**3.51**	3.74	8.55	9.67	8.58	**8.55**	urls
9.19	5.78	**5.04**	5.15	14.68	15.23	13.85	**13.74**	chinese
8.22	5.27	**4.69**	5.06	14.44	14.63	14.50	**14.47**	japanese
4.70	3.11	**2.77**	3.07	9.45	9.11	9.44	**9.14**	ps_comment
8.21	5.27	**4.46**	4.87	14.74	12.66	14.30	**12.65**	wikipedia
7.38	4.75	**4.11**	4.32	12.27	10.83	11.14	**10.32**	

6.3 Encoding Kernels

Comparing variant 7 and 8 in Table 3, we see that AVX512 improves encoding performance by 2.5×. Table 2 investigates the performance of different encoding kernel implementations. The *simd* columns show the encoding performance of the SIMD kernel with different loop unrolling counts. The best performance is achieved with 3-way unrolling (i.e., "simd$_3$"). The remaining columns of Table 2 show different scalar encoding kernels. Using histogram statistics on the symbol table, our implementation adaptively chooses the best kernel automatically.

6.4 Evolution of FSST

The FSST compression algorithm went through several iterations before arriving at the current design. This evolution is retraced in Table 3, which shows the compression factor (CF), symbol table construction cost (CS), and string encoding speed (SE) for 8 variants. As mentioned earlier, our first design was based on a suffix array and achieved a respectable compression factor of 1.97×, but requires 74.3 cycles/byte for symbol table construction and 160 cycles/byte for encoding. Our current AVX512 version (variant 8 in the figure) is 90× faster for table construction and 40× faster for encoding than the first version – while providing a higher compression factor. The final variant is also much faster than the initial FSST version (variant 4), due to lossy hashing, fewer iterations, adaptive kernel selection, and AVX512

Table 3: Evolution of FSST compression techniques – top to bottom. Properties in terms of compression factor (CF), symbol table construction (SC) cost in cycles-per-byte, when constructing a new symbol table for each 8MB of text, and string encoding (SE) cost cycles-per-byte

variant 1: suffix-array based construction (slowest)		
symbol table org:	sorted	CF: **1.97**
string encoding:	dynamic programming	SC: 74.3cyc/b
symbol matching:	strncmp	SE: 160.0cyc/b
variant 2: suffix-array based construction (slow)		
symbol table org:	sorted	CF: **1.97**
string encoding:	dynamic programming	SC: 74.3cyc/b
symbol matching:	str-as-long	SE: 81.6cyc/b
variant 3: suffix-array based construction (less slow)		
symbol table org:	sorted	CF: **1.94**
string encoding:	greedy	SC: 74.3cyc/b
symbol matching:	str-as-long	SE: 37.4cyc/b
variant 4: FSST - initial idea		
symbol table org:	sorted	CF: **2.33**
string encoding:	greedy	SC: 2.1cyc/b
symbol matching:	str-as-long	SE: 20.0cyc/b
variant 5: FSST - lossy-perfect-hash		
symbol table org:	lossy perfect hash	CF: **2.28**
string encoding:	greedy (branchy)	SC: 1.85cyc/b
symbol matching:	str-as-long	SE: 12.4cyc/b
variant 6: FSST - optimized construction		
symbol table org:	lossy perfect hash	CF: **2.19**
string encoding:	greedy (branchy)	SC: 0.82cyc/b
symbol matching:	str-as-long	SE: 11.3cyc/b
variant 7: FSST - adaptive kernel i.e., min($short, single, hash$)		
symbol table org:	lossy perfect hash	CF: **2.19**
string encoding:	greedy (adaptive)	SC: 0.82cyc/b
symbol matching:	str-as-long	SE: 10.32cyc/b
variant 8: FSST - AVX512 kernel 3-way unrolled ($simd_3$)		
symbol table org:	lossy perfect hash	CF: **2.19**
string encoding:	greedy (predicated)	SC: 0.82cyc/b
symbol matching:	AVX512	SE: 4.11cyc/b

– though we had to sacrifice some the space gains relative to variant 4. The numbers in Table 3 also show that symbol table construction time is a fraction of encoding time, despite requiring multiple iterations. This is because symbol table construction works on a sample rather than the full data set.

7. SUMMARY AND FUTURE WORK

Fast Static Symbol Table (FSST) is a lightweight compression scheme for strings that exploits frequently-occurring substrings. We present here fast algorithms for decompression and compression. For textual data, FSST on average achieves compression factors of over 2×, compresses with almost 1GB/s using AVX512, and decompresses with 1.75GB/s. FSST thereby outperforms the widely-used LZ4 compression method on these three metrics. However, in contrast to LZ4, FSST also support random access to individual strings without having to decompress a block of data. This makes FSST particularly useful for database systems, which can exploit random access, for example, during index lookups. In the future, we investigate which operations besides equality can be performed directly on FSST-compressed strings without having to decompress them first.

8. REFERENCES

[1] http://bit.ly/2uEKhzJ (shortened URI). Full URI: https://web.archive.org/web/20200229161007/https://www.percona.com/blog/2016/04/13/evaluating-database-compression-methods-update/.

[2] http://bit.ly/3adzJXu (shortened URI). Full URI: https://web.archive.org/web/20200229161613/https://www.postgresql.org/docs/11/storage-toast.html.

[3] J. Arz and J. Fischer. Lempel-Ziv-78 compressed string dictionaries. *Algorithmica*, 80(7):2012–2047, 2018.

[4] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, pages 283–296, 2009.

[5] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In *SIGMOD*, pages 271–282, 2001.

[6] P. Damme, D. Habich, J. Hildebrandt, and W. Lehner. Lightweight data compression algorithms: An experimental survey. In *EDBT*, pages 72–83, 2017.

[7] P. Gage. A new algorithm for data compression. *C Users J.*, 12(2):23–38, Feb. 1994.

[8] B. Ghita, D. G. Tomé, and P. A. Boncz. White-box compression: Learning and exploiting compact table representations. In *CIDR*, 2020.

[9] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *SIGMOD*, pages 389–400, 2007.

[10] S. Jain, D. Moritz, D. Halperin, B. Howe, and E. Lazowska. SQLShare: Results from a multi-year SQL-as-a-service experiment. In *SIGMOD*, pages 281–293, 2016.

[11] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper. Data Blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *SIGMOD*, pages 311–326, 2016.

[12] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Data Compression Conference*, pages 296–305, 1999.

[13] R. Lasch, I. Oukid, R. Dementiev, N. May, S. Demirsoy, and K.-U. Sattler. Fast & strong: The case of compressed string dictionaries on modern cpus. In *Damon*, 2019.

[14] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[15] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.*, 45(1):1–29, 2015.

[16] I. Müller, C. Ratsch, and F. Färber. Adaptive string dictionary compression in in-memory column-store database systems. In *EDBT*, pages 283–294, 2014.

[17] B. Raducanu, P. A. Boncz, and M. Zukowski. Micro adaptivity in vectorwise. In K. A. Ross, D. Srivastava, and D. Papadias, editors, *SIGMOD*, pages 1231–1242, 2013.

[18] V. Raman, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Müller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. J. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[19] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Muehlbauer, T. Neumann, and M. Then. Get real: How benchmarks fail to represent the real world. In *DBTEST*, 2018.

[20] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, 2000.

[21] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes (2nd Ed.): Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

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

[23] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, 2006.