FSST: Fast Random Access String Compression

Peter Boncz
CWI
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 Access String Compression. *PVLDB*, 13(11): xxxx-yyyy, 2020.
DOI: https://doi.org/10.14778/3407790.3407851

1. INTRODUCTION

In many real-world databases, including ERP [18] and visual analytics [21], 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, and many systems rely on dictionaries to compress strings. 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, many systems apply dictionary compression on limited chunks or blocks of data rather than the entire relation, which may limit its effectiveness further.

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407851

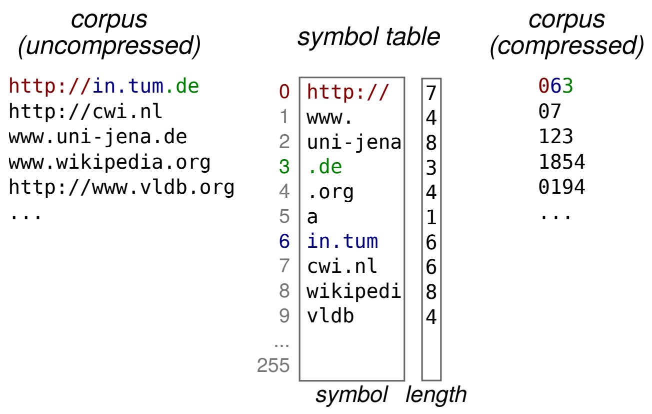

Figure 1: FSST provides very fast – yet effective – string compression, by replacing *symbols* of length 1-8 by 1-byte *codes*. Finding a good *symbol table* is a key challenge, addressed in Section 4. Techniques to make FSST compression very fast, including AVX512, are described in Section 5.

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 are not suited for compressing small, individual strings, because they require input sizes on the order of several kilobytes for good compression factors (cf. Section 6.1). 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 [9, 14, 4, 15] are much slower than FSST on bulk compression and decompression, which 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 ($\approx$ 1-3 cycles/byte, or 1-3 GB/s per 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 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, 2]), FSST is better over all dimensions. FSST provides comparable – but often, faster – decompression and compression speed, and noticeably 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 [18, 21], this can have a substantial impact on overall space consumption. The source code of FSST under the MIT License, the "dbtext" compression database text corpus we contribute, and the replication package of this paper are available:

`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 and present future work in Section 7.

2. RELATED WORK

Most research on lightweight compression for database systems concentrates on integer data [25, 11, 13, 20, 17, 8]. Similarly, work on query processing on compressed data generally does not focus on strings [22, 6]. We argue that given the prevalence and performance challenges of strings in real-world workloads [18, 12, 21], more research is required.

The most common approach for compressing strings is de-duplication using dictionaries [25, 11, 13, 20]. 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. [5] 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 [4], who developed a variant of LZ78 [24] 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 [4]. 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" [3], 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 [9, 23] 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 [14] (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 [15], 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 one without escaping thanks to its higher compression factor.

Our open-source implementation optimizes decoding by detecting the *absence* of escapes (byte 255) in the next 4-byte word using computation, and if so, decodes 4 codes without having to look for escapes. It handles the *presence* of escapes efficiently using a programming trick called "Duff's device". All in all, FSST decoding is among the fastest string decompressors, approaching 2GB/s in our evaluation.

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 4.3.

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 a FSST-compressed column is queried, 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. 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 is often significantly less.

Our first attempt at symbol table construction created a suffix array to identify the symbols with highest gain. With a suffix array-based approach, the first symbol picked will indeed have the highest compression gain. However, the compression gain of subsequent symbols depends on earlier symbols. Correcting for the dependencies on earlier symbols is very difficult and, depending on how it is done, leads to large over- or underestimates. For this reason this approach produced significantly worse symbol tables than the evolutionary algorithm we will present subsequently.

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

[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.

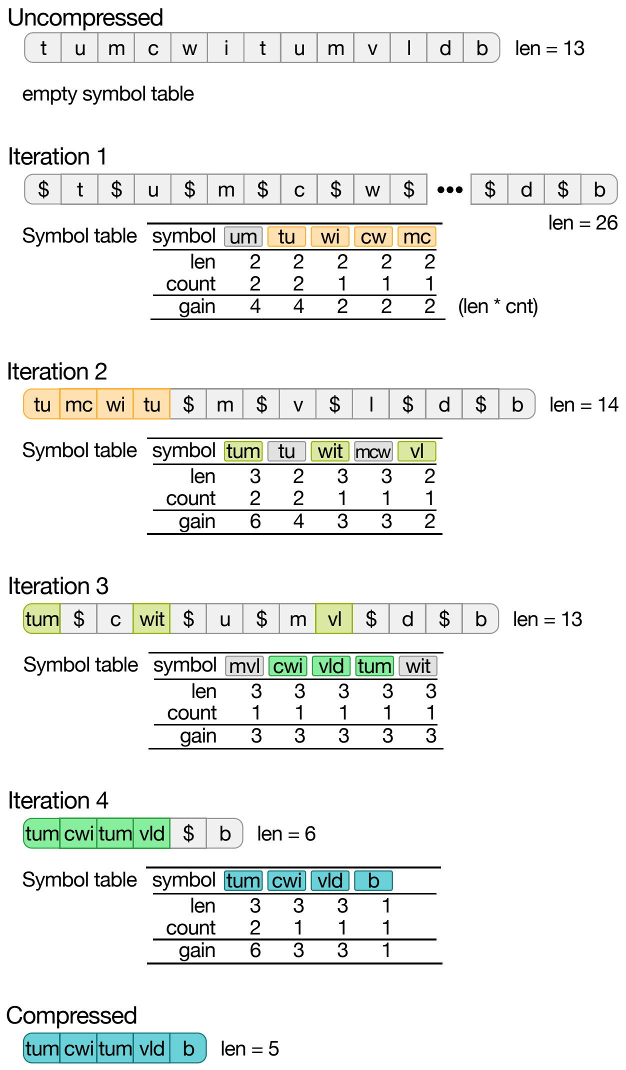

Figure 2: Four iterations of symbol table construction algorithm on the corpus "tumcwitumvldb" with a maximum symbol length of 3 and a maximum symbol table size of 5. "$" stands for the escape byte.

expensive. The number of possible symbols 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.

4.2 Algorithm Overview

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[2].

Figure 2 illustrates the algorithm by showing 4 iterations on the example corpus "tumcwitumvldb". To keep the example manageable, we limit the maximum symbol length to 3 (rather than 8) and that the maximum symbol table size to 5 (rather than 255). After each 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. 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. 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". The former two of these top symbols ("um", "tu") have a gain of 4 since they occur twice, while the latter three symbols ("wi", "cw", and "mc") occur just once and therefore have a gain of 2. Note that the symbols "mv", "vl", "ld", "db", "m", "t", "u" also have a gain of 2 and could have been picked as well. In other words, when picking the top symbols, the algorithm resolved ties arbitrarily.

[2]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.

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 Bottom-Up Symbol Table Construction

Algorithm 3 shows pseudo-code in Python style for the bottom-up algorithm. The open-source C++ FSST implementation is of course much faster. Nevertheless, class, method, and variable names in this pseudo-code follow our real implementation. The main methods are:

- `buildSymbolTable()`. This is the top-level entry point of the algorithm. Given a `text`, it builds a symbol table in 5 iterations; starting with an empty `symbolTable()`. In it, the `st.symbols[]` array always starts with 256 pseudo-symbols, representing single bytes. These are used to represent and administer the frequency of escaped bytes (i.e., the situation where compression would use 2 bytes to represent a byte of text due to an extra escape byte 255 getting generated). The next `st.nSymbols`, up to 255, entries in the `st.symbols[]` array contain the real symbols (initially `st.nSymbols=0`).
- `compressCount()`. This method compresses a text using the current symbol table `st`. In the first iteration with the empty symbol table, it will only use escaped bytes and the result will be twice the input size. Rather than producing compressed output text, this method just records the frequency of the codes or bytes it encounters; as well as the frequencies of the *subsequent* codes or bytes, in two arrays (`count1[]` and `count2[][]`, where the latter is 2-dimensional).
- `makeTable()`. Using the frequencies in `count2[][]` and `count1[]`, it generates all possible candidate symbols and calculates their *gain*. It considers all single-bytes as new symbols (`st.symbol[0..255]`), all symbols of the previous generation (`st.symbol[256..256+st.nSymbols]`), but also all combinations of these (concatenations, up to length 8). Using a priority queue, the 255 symbols with highest *apparent gain* (which is length*count) are inserted in the new symbol table.
- `findLongestSymbol()`. This method finds the longest matching (pseudo-)symbol in a `text`. We store the (real) symbols (those from code 256 on) in lexicographical order, but when one string prefixes the other, the longest is first. This means that when testing for prefix match all real symbols from first to last, the first hit will be the longest. This method restricts the search to the range of symbols that start with the first byte `text[0]`. For this purpose, there is an `st.sIndex[byte]` array that keeps the position of the first (real) symbol that starts with a certain byte.
- `makeIndex()`. This helper method is called to finalize a new symbol table. It sorts the symbols lexicographically as described above and initializes the `st.sIndex[]`.

Algorithm 3 Simplified Python-style pseudo code for bottom-up symbol table construction.

```
class SymbolTable:
  def __init__(st): # constructor
    st.nSymbols = 0
    st.sIndex[257] = [0]*256
    st.symbols[512] = ['']*512
    # the first 256 symbols are escaped bytes
    for code in range(0,255)
      st.symbols[code] = chr(code)

  def insert(st, s):
    st.symbols[256+st.nSymbols++] = s

  def findLongestSymbol(st, text):
    var letter = ord(text[0])
    # try all symbols that start with this letter
    for code in range(st.sIndex[letter],st.sIndex[letter+1])
      if (text.startswith(st.symbols[code]))
        return code # symbol, code >= 256
    return letter # non-symbol byte (will be escaped)

  # compress the sample and count the frequencies
  def compressCount(st, count1, count2, text):
    var pos = 0
    var prev, code = st.findLongestSymbol(text[pos:])
    while ((pos += st.symbols[code].len()) < text.len())
      prev = code
      code = st.findLongestSymbol(text[pos:])
      # count the frequencies
      count1[code]++ # count single symbol[code]
      count2[prev][code]++ # count concat(prev,code)
      # we also count frequencies for the next byte only
      if (code >= 256)
        nextByte = ord(text[pos])
        count1[nextByte]++
        count2[prev][nextByte]++

  def makeTable(st, count1, count2): # pick top symbols
    var res = SymbolTable()
    var cands = []
    for code1 in range(0,256+st.nSymbols)
      # single symbols (+all bytes 0..255) are candidates
      gain = st.symbols[code1].len() * count1[code1]
      heapq.heappush(cands, (gain, st.symbols[code1]))
      for code2 in range(0,256+st.nSymbols)
        # concatenated symbols are also candidates
        s = (st.symbols[code1]+ st.symbols[code2])[:8]
        gain = s.len() * count2[code1][code2]
        heapq.heappush(cands, (gain, s))
    # fill with the most worthwhile candidates
    while (res.nSymbols < 255)
      res.insert(heapq.heappop(cands))
    return res.makeIndex()

  def makeIndex(st): # make index for findLongestSymbol
    # sort the real symbols and init the letter index
    var tmp = sort(st.symbols[256,256+st.nSymbols])
    for i in range(0,st.nSymbols).reverse()
      var letter = ord(tmp[i][0])
      st.sIndex[letter] = 256+i
      st.symbols[256+i] = tmp[i]
    st.sIndex[256] = 256+st.nSymbols # sentinel
    return st

  def buildSymbolTable(st, text): # top-level entry point
    var res = SymbolTable()
    for generation in [1,2,3,4,5]
      var count1[512] = [0]*512
      var count2[512][512] = [count1]*512
      st.compressCount(res, count1, count2, text)
      res = st.makeTable(res, count1, count2)
    return res
```

4.4 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 factors. We observed that 5 iterations are generally enough to converge to a good compression factor.

Besides the number of iterations, symbol table construction speed obviously also depends on the size of training corpus. Luckily, we can train our algorithm using a sample rather than using the full corpus: Intuitively, using a sample 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 to 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. We also reduce the size of the sample *adaptively*: growing it from 6% to 100% of the full sample linearly over the 5 iterations. Finally, to improve cache efficiency, we split the 256K `count2[][]` counters into 4 minor bits (frequently accessed) and 12 (infrequently accessed) high bits.

5. OPTIMIZING COMPRESSION SPEED

The performance-critical method for compression in Algorithm 3 is `findLongestSymbol()`. Our eventual goal is very high compression performance using SIMD. An important restriction in SIMD is that loops and branches are not supported. However, `findLongestSymbol()` *loops* over all symbols that start with a particular byte, and compares the strings (`startswith()`) and *branches* away on the first hit.

We therefore eliminate the use of loops and branches from the scalar code in Section 5.1 and describe its AVX512 version in Section 5.2. First, however, we describe the data structures needed for this.

Lossy Perfect Hashing. 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 hash 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.

Algorithm 4 Lossy Perfect Hashing: no loops & branches

```
struct Symbol {
  union val { char buf[8]; uint64_t num}; // allows2compare str as int
  uint16_t code; // bits [0..8]=code [12..15]=len. Unused: code=511
  uint16_t ignoredBits; // unused bits in num, i.e. 64-len*8
}

struct SymbolTable {
  uint8_t nSymbols; // # of normal symbols (not counting escapes)
  Symbol symbols[512]; // all symbols: 0-255 escapes, then n Symbols

  // uint16_t stores code&length: resp. bits [0..8] and bits [12..15]
  uint16_t shortCodes[256][256];//codes (511=unused) of 1-2byte symbs

  Symbol hashTab[hashTabSize]; // keyed on the first three bytes
  static uint64_t hashTabSize = 4096; // fits L1
  uint64_t hash(uint64_t x) { return (x*2971215073)^(x>>15); }
}

void encodeScalar(uint8_t*& cur, uint8_t*& out, SymbolTable& st){
  uint64_t word = *(uint64_t*)cur;

  // speculatively write 1st byte (required for escapes, else harmless)
  out[1] = (uint8_t) word;

  // lookup in lossy perfect hash table
  uint64_t idx = hash(word & 0xFFFFFF) & (st.hashTabSize-1);
  Symbol s = st.hashTab[idx]; // fetch symbol from hash table
  uint64_t num = word & (0xFFFFFFFFFFFFFFFF >> s.ignoredBits);

  uint16_t code = (s.val.num==num & s.code!=511) ? // hastable hit?
           s.code : st.shortCodes[word&0xFFFF]; // conditional move
  out[0] = (uint8_t) code; // write out code. Note: (uint8_t) 511=255

  // advance the pointers with predication (i.e. without branches)
  out += 2-((code>>8)&1); // increase with 1 or 2 (escape = 9th bit)
  cur += (code>>12); // symbol length is in bits [12..15] of code
}
```

The additional `shortCodes[A][B]` array has 65536 entries (`A` and `B` are bytes) used to check whether there is a 2-byte symbol `AB` 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. After inserting the 2-byte codes in the array, we put in *all* free slots `shortCodes[A][*]` the code of 1-byte symbols `A`. Thus, we can use the array to check whether either a 2- or 1-byte symbol matches.

The optimized `findLongestSymbol()` first checks a string prefix match with the symbol in the perfect hash table found by using the next 3 bytes `ABC` as lookup key, and fetches code `X` of that symbol. If there is a `hit`, a symbol of length 3 or more matches the text. Otherwise, it fetches `Y = shortCodes[A][B]` as the next code. The choice between these two can be made using a *conditional move* (hit?X:Y), which is also supported in SIMD. This way, no loop or branch is needed.

5.1 Predicated Scalar Compression.

In Algorithm 4 we show a scalar FSST-compression kernel that advances one symbol in a text, and effectively inlines `findLongestSymbol()`. The data layout is quite optimized: codes (0-511) are represented as 16-bit integers, but we also store the length of the symbol in its 4 highest bits (bits 12-15). We leverage the fact that FSST symbols fit into a 8-byte word to avoid string comparisons. This can be seen in the `union` C++ definition of `Symbol` in the first lines of Algorithm 4. We even pre-materialize the amount of unused bits in symbols shorter than 8 (i.e., 64-8*len) as `ignoredBits` to speed up the hash lookup by two fewer operations.

The current position in the text is `cur`, and the compressed string is appended at `out`; both these in-out parameters are

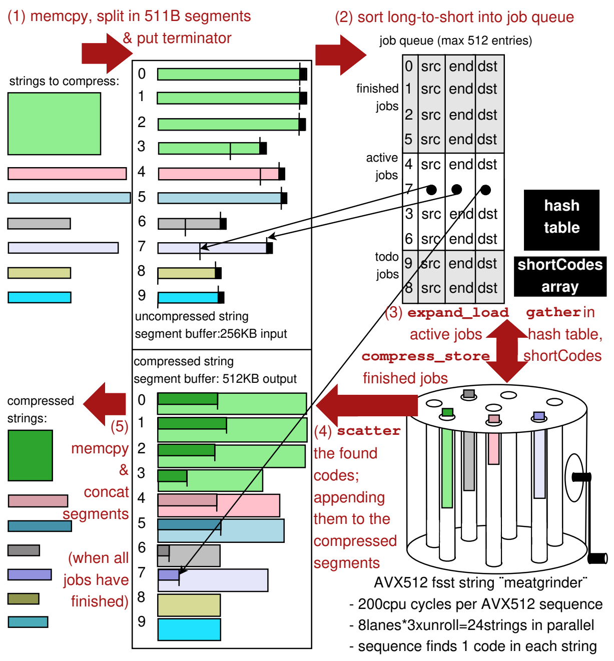
(1) memcpy, split in 511B segments
& put terminator
(2) sort long-to-short into job queue
job queue (max 512 entries)
strings to compress:
finished jobs
active jobs
todo jobs
0 src end dst
1 src end dst
2 src end dst
5 src end dst
4 src end dst
7
3 src end dst
6 src end dst
9 src end dst
8 src end dst
hash table
shortCodes array
uncompressed string segment buffer:256KB input
compressed string segment buffer: 512KB output
(3) expand_load active jobs
compress_store finished jobs
gather in hash table, shortCodes
compressed strings:
(5) memcpy & concat segments
(when all jobs have finished)
(4) scatter the found codes; appending them to the compressed segments
AVX512 fsst string "meatgrinder"
- 200cpu cycles per AVX512 sequence
- 8lanes*3xunroll=24strings in parallel
- sequence finds 1 code in each string

Figure 3: AVX-512 FSST compression: a "meat-grinder" that encodes 24 strings in parallel

moved forward. While we eventually write one byte to `out[0]` (which due to the cast-to-byte will be the escape symbol 255 if the `code=511`, i.e. no symbol found), earlier we *speculatively* wrote the current byte to `out[1]`. This speculative write handles the case that FSST needs to escape a byte, but is harmless in case a real code is found.

All in all, Algorithm 4 demonstrates a FSST compression kernel without loop or branch. It can compress strings at 10cycles per byte, which is 400MB/s on our platform, putting it among the fastest string compressors already.

An issue that we glossed over so far is dealing with end-of-string correctly. This kernel can jump over end-of-string, as it blindly loads the next 8 input bytes into `word`. 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 (step 1 in Figure 3).

5.2 Compression in AVX512

The FSST API compresses a batch of strings, preferably 100 or more. It is also efficient with fewer strings, even just one, 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, and appending the terminator. This is shown in step (1) of Figure 3. When 512 segments have been gathered or there is no more data, the AVX512 encoding kernel in Algorithm 5 is invoked. Each segment is an *encoding job*. The while-loop in Algorithm 5 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, times 3 due to unrolling). We call this a "meat-grinder", as we push data through a compression cylinder (AVX512 kernel) with 24 strings in parallel.

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 3× 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, 3× 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. Retiring jobs (lanes in the job control register) uses the `compress_store` instruction, and refilling the `expand_load` instruction, as depicted in step (3) of Figure 3. In step (2) of Figure 3 we first radix-sorted the job queue 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 920MB/s. As such, FSST is the fastest known string compressor available.

This AVX512 encoding kernel led to slight adaptations of the FSST format. Not only the SIMD, but also the scalar code uses 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. Further, the scalar method also uses the terminator byte as a fast way to avoid going over end-of-string. The terminator byte is meta-data

Algorithm 5 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, mask, num;
   __mmask8 hit, 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*12 (bucket width)
      icl = _mm512_i64gather_epi32(idx,hashTab,1);//probe hash table
      // icl (i,c,l) = uint32 ignoredBits:16,code:12,len:4

      // fetch the symbol (text) part of the hash table record and compare
      num = _mm512_i64gather_epi64(idx,hashTab+8,1);//next 8bytes
      hit = _mm512_cmplt_epi64_mask(icl, x8_FF0000); // used?
      mask = _mm512_and_epi64(icl,x8_FF); // get ignoredBits
      mask = _mm512_srlv_epi64(x8_FFFFFFFFFFFFFFFF, mask);
      word = _mm512_and_epi64(word, mask); // clean the word with mask
      hit &= _mm512_cmpeq_epi64_mask(num, word); // hit?
      icl = _mm512_srli_epi64(icl, 16); // extract code+len from icl

      // conditional move: select between shortCodes and hashTab (hit)
      code = _mm512_mask_mov_epi64(code, hit, 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_sub_epi64(x8_2, 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
}
```

that is added to the symbol table, such that 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).

6. EVALUATION

FSST has been designed for compressing textual string columns. To evaluate it, we curated a text corpus (dubbed "dbtext") consisting of 23 string columns covering a wide variety of real-world string data. The columns, which we believe to be typical for database text attributes, 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 such as Wikipedia, Tableau Public [10], or IMDb [16]. For reproducibility, the data sets have been published together with the MIT-licensed 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 [21].

All experiments were performed on a workstation with 32GB RAM and a single 10-core (20-hyperthread) 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).

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 2.29×. Thus, as a rule of thumb, FSST halves space utilization for textual data. LZ4, for comparison, achieves an average compression factor of 1.70×.

Table 1 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

Table 1: Evaluation data sets ("dbtext" corpus) and performance of FSST versus LZ4 in terms of compression factor, compression speed, and decompression speed. Each data set is treated as a 8MB file.

	avg		compr. factor		compr. [MB/s]		decompr. [MB/s]	
name	len	example string	LZ4	FSST	LZ4	FSST	LZ4	FSST
hex	8	DD5AF484	1.14	**2.11**	**1,043**	888	**1,844**	1,455
yago	19	Ralph_A._Brown	1.25	**1.63**	572	**753**	**1,394**	1,052
email	22	xnj_14@hotmail.com	1.55	**2.13**	627	**932**	**2,393**	1,519
wiki	23	Benzil	1.31	**1.63**	556	**738**	**1,489**	1,159
uuid	37	84e22ac0-2da5-11e8-9d15- ...	1.55	**2.44**	637	**1,075**	**2,792**	2,549
urls2	55	http://fr.wikipedia.org/ ...	1.75	**2.05**	601	**909**	**2,173**	2,051
urls	63	http://reference.data.go ...	**2.77**	2.42	711	**1,042**	**2,253**	2,229
firstname	7	RUSSEL	1.25	**2.04**	606	**775**	**868**	816
lastname	10	BALONIER	1.28	**1.97**	585	**819**	**969**	965
city	10	ROELAND PARK	1.37	**2.14**	519	**856**	1,049	**1,064**
credentials	11	PHD, HSPP	1.48	**2.31**	442	**912**	1,060	**1,151**
street	13	PURITAN AVENUE	1.60	**2.35**	502	**974**	**1,284**	1,271
movies	21	Return to 'Giant'	1.23	**1.66**	542	**784**	**1,446**	1,109
faust	24	Erleuchte mein bedürftig Herz.	1.48	**1.87**	422	**817**	1,377	**1,477**
hamlet	30	<LINE>That to Laertes ...	2.13	**2.41**	549	**1,022**	1,772	**2,120**
chinese	87	道人决心消除肉会 ...	1.40	**1.69**	543	**778**	**2,507**	2,076
japanese	90	せん。しかし、...	1.84	**2.00**	463	**891**	2,544	**2,558**
wikipedia	130	Weniger häufig fressen sie ...	1.45	**1.81**	518	**820**	**2,642**	2,330
genome	10	atagtgaag	1.59	**3.32**	536	**1,203**	1,080	**2,671**
location	40	(40.84242764486843, -73 ...	1.58	**2.51**	433	**1,098**	2,279	**2,548**
c_name	19	Customer#000010485	3.08	**3.80**	1,140	**1,443**	2,982	**3,404**
l_comment	27	nal braids nag carefully expres	2.22	**2.90**	608	**1,193**	1,665	**1,809**
ps_comment	124	c foxes. fluffily ironic ...	2.79	**3.40**	742	**1,342**	2,709	**3,799**
		average	1.70	**2.29**	604	**959**	1,851	**1,877**

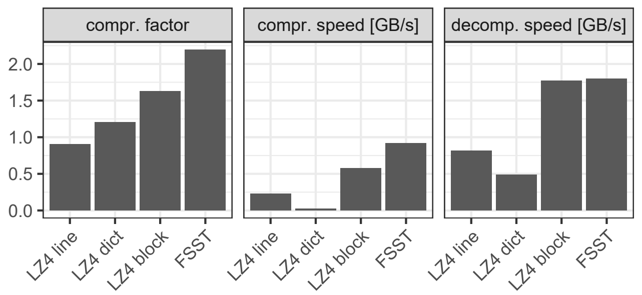

Figure 4: With LZ4 short strings do not compress well, even with a pre-generated dictionary.

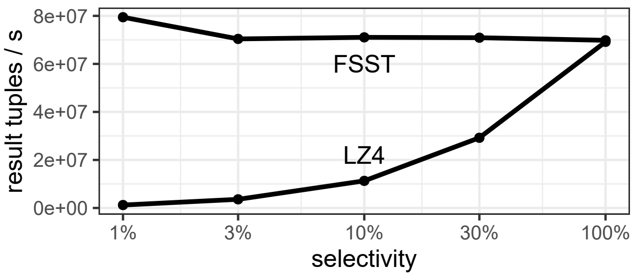

Figure 5: Selective queries are fast in FSST due to random access to individual values.

terms of the compression factor and compression speed. On average, besides resulting in a 34% better compression factor[3], 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%).

So far, we treated each data set as one 8MB file, which works well with block-based approaches like LZ4. To understand whether LZ4 would also work on smaller block sizes, we split the *urls* data set into blocks of different sizes and compressed each one individually:

blocksize (bytes)	64K	16K	4K	1K	256	64	16
compr. factor	2.73	2.45	2.03	1.59	1.14	0.78	0.46

[3]If one fully sorts the lines in the files of the dbtext corpus lexicographically, LZ4 compression improves by 15%; still not catching FSST, which is indifferent to the very localized text similarities such sorting creates.

This shows that LZ4 compression suffers from blocks <27KB and that at least a few kilobytes are needed to achieve reasonable compression. String sizes of <100 bytes, common in databases, result in larger data sizes.

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.

Table 2: Detailed FSST encoding performance as cycles-per-input-byte for various encoding kernels. "simd$_3$" is fastest, i.e., AVX512 using 3-way enrolling of the kernel in Algorithm 4. It is 2.5x faster than scalar encoding.

$simd_1$	$simd_2$	$simd_3$	$simd_4$	$scalar$	
8.01	5.36	**4.98**	5.16	**11.26**	firstname
7.97	5.08	**4.17**	4.42	**5.09**	hex
8.70	5.51	**5.07**	4.69	**10.82**	city
5.29	3.57	**2.85**	3.26	**9.00**	genome
8.05	5.12	**4.18**	4.44	**10.35**	lastname
7.82	5.03	**4.68**	4.61	**13.41**	credentials
7.35	5.02	**4.36**	4.52	**11.46**	street
3.87	2.76	**2.23**	2.42	**5.37**	c_name
9.36	5.91	**5.03**	5.31	**11.50**	yago
9.10	5.74	**4.82**	5.14	**11.36**	movies
7.19	4.60	**4.03**	4.25	**10.13**	email
9.37	5.92	**4.93**	5.30	**11.88**	wiki
8.55	5.51	**4.98**	5.18	**13.05**	faust
5.59	3.63	**3.16**	3.42	**9.78**	l_comment
7.35	4.56	**4.13**	4.23	**10.50**	hamlet
6.36	4.08	**3.40**	3.59	**8.06**	uuid
5.64	3.70	**3.05**	3.14	**5.73**	location
7.50	4.72	**4.03**	4.28	**10.12**	urls2
6.32	4.04	**3.51**	3.74	**8.55**	urls
9.19	5.78	**5.04**	5.15	**13.74**	chinese
8.22	5.27	**4.69**	5.06	**14.47**	japanese
4.70	3.11	**2.77**	3.07	**9.14**	ps_comment
8.21	5.27	**4.46**	4.87	**12.65**	wikipedia
7.38	4.75	**4.11**	4.32	**10.32**	

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.

6.3 Non-textual data

Outside our database context, the compression community often evaluates compression methods on the Silesia corpus, which consists of 11 files, of which 4 are textual (dickens, reymont, mr, webster), one is XML and 6 are binary. FSST achieves 10% better compression sizes on the text files, but is 25% worse on the binaries, on average.

While we think binary files are not relevant for FSST, its compression ratio on large XML and JSON files, which are relevant, is 2-2.5x worse than LZ4. However, we think database systems should store these composite values not as simple strings, but in a specialized type that allows query processing. For instance, Snowflake recognizes the structure in JSON columns, and internally stores each often-occurring JSON attribute in a separate internal column [7].

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 (predicated)	SC: 1.85cyc/b
symbol matching: str-as-long	SE: 10.32cyc/b
variant 6: FSST - optimized construction	
symbol table org: lossy perfect hash	CF: 2.19
string encoding: greedy (predicated)	SC: **0.82cyc/b**
symbol matching: str-as-long	SE: 10.32cyc/b
variant 7: 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**

This means that attribute names are not repetitively stored, saving space, and only their values are stored in an appropriately typed internal column. In case of strings, such internal columns could then be compressed with FSST.

6.4 Encoding Kernels

Comparing variant 6 and 7 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. Best performance is achieved with 3-way unrolling (i.e., "simd$_3$"), beating scalar encoding by a factor 2.5.

6.5 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 7 variants. As mentioned earlier, our first design was based on a suffix array and achieved a respectable compression factor of 1.97×, but required 74.3 cycles/byte for symbol table construction and 160 cycles/byte for encoding. Our current AVX512 version (variant 7 in the figure) is 90× faster for table construction and 40× faster

Table 4: Query execution times in ms for TPC-H SF10 with compressed string columns, using 20 threads.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22	geo. mean
uncompressed	118	14	71	44	54	25	93	40	186	75	15	54	228	33	27	65	26	165	99	38	120	49	57
LZ4	131	23	72	48	53	25	80	42	190	105	15	55	250	34	27	64	26	166	91	40	118	50	59
FSST	104	15	69	42	53	25	85	39	186	71	15	52	235	29	27	52	25	162	69	39	115	28	53

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), thanks to lossy perfect hashing and AVX512 – though we had to sacrifice about 6% of the space gains relative to variant 4. Table 3 also shows that symbol table construction is only a fraction of encoding time, despite requiring multiple iterations. Optimizing contruction entailed reducing iterations from 10 to 5, building on a sample (that grows every iteration), and shrinking the memory footprint of the counters.

6.6 System Integration and Query Processing

To measure some end-to-end effects of FSST on query processing, we did a prototype integration in our Umbra system [19]. There are different ways how one could integrate FSST into a database system. Our implementation compresses each string individually, and then decompresses it as late as possible. Equality predicates against a constant can be evaluated directly upon the compressed form and decompression is needed just for non-equality comparisons, sorting, and result printing. We also created an LZ4 integration in Umbra, where we organize tuples in blocks of size 2^{16} and then compress all string values of a given column in that block. There, random access is no longer possible, and strings have to be decompressed on block before access. Block decompression is triggered only when needed, i.e., only if there are qualifying tuples after checking the non-string predicates.

Table 4 shows TPC-H results on SF10, using 20 threads. As expected, compression had little effect, as TPC-H is usually dominated by joins on integer columns. There are some notable exceptions though. Q13 is dominated by a `like` predicate on o_comment, and thus directly shows the overhead of decompression. Both FSST and LZ4 are very fast here, with a slow-down of only 3% resp. 9%. Q19 makes heavy use of well-compressible string columns, and performance in fact improves. That is especially true for FSST, as it not only saves scan memory bandwidth, but also allows to push down string filter predicates.

In terms of space, the size of the string pool is 4.1GB uncompressed, 1.5GB with LZ4, and only 0.69GB with FSST. The compression factor of FSST is inflated by the fact that Umbra inlines short strings of 12 bytes or less, and thereby often avoids allocation in the string pool. Many TPC-H strings happen to fall below that threshold after compression. Space consumption differences are likely less strong under other circumstances.

In a different TPC-H experiment, Müller et al. [18] replaced all TPC-H (integer) key columns by strings. Replicating this artificial experiment would not have the desired effect in Umbra. To allow for random access, our string values are split into a fixed-size header and a variable part that is referenced by the header. For short strings our system directly inlines the string into the header, and only stores a pointer value for longer strings. Storing the integer keys as text would result in short strings only, which would effectively disable compression. To replicate the spirit of their experiment, we prepended the string "keyvalue-padded" before every key value, which makes the key columns sufficiently large to see compression effects.

When running the query `SELECT COUNT(*) FROM orders, lineitem WHERE o_orderkey = l_orderkey`
we now get query execution times of 484ms uncompressed, 560ms when using LZ4, and 554ms when using FSST. The overhead is not caused by the decompression (which is necessary because the two join predicate columns use different dictionaries) but by the effects on the rest of the system: in the uncompressed case, the strings are known to be stable, and the system just stores them as they are. When building a hash table for decompressed strings, the system has to make a copy, as the decompressed value will go away. A profiler run shows that the overhead is largely caused by memcpy into the hash table, whereas the FSST decompression itself takes just 9% of the time.

To summarize, the overhead of adding FSST compression on TPC-H is small. Even in the worst-case experiment with padded strings as key columns the overhead is 14%. For more realistic scenarios, where just string columns are compressed the overhead is at most 3%, and queries like Q19 we in fact become faster by 30% by adding compression. TPC-H has few selective predicates, thus FSST cannot show off its random access capabilities here, but even in this bulk-processing scenario it outperforms LZ4.

7. SUMMARY AND FUTURE WORK

Fast Static Symbol Table (FSST) is a lightweight, random access compression scheme for strings that exploits frequently-occurring substrings in a column. We presented fast algorithms for decompression and compression. For textual data, FSST on average achieves compression factors of over 2×, compresses at 4 cycles per byte with AVX512, and decompresses at 2 cycles per byte (resp. 1GB/s and 2GB/s on our platform). FSST thereby outperforms even the heavily optimized LZ4 compression library on these three metrics. However, in contrast to LZ4, FSST also supports efficient 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. Beyond database systems, we also envision applications in information retrieval, network/cloud storage, text analysis and more.

In the future, we will investigate which operations besides equality can be performed directly on compressed strings without having to decompress them first. For example, we believe it is possible to develop a Knuth–Morris–Pratt-like substring search algorithm directly operating on FSST-compressed strings. It would further be interesting to explore a hardware implementation of FSST decompression. The small size of symbol table, the fact that it is static, and finally the simplicity of the decompression algorithm, make such an undertaking highly feasible.

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/3ajy0QD (shortened URI). Full URI: https://web.archive.org/web/20200229154849/https:/github.com/inikep/lzbench.

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

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

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

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

[7] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. The snowflake elastic data warehouse. In *SIGMOD*, 2016.

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

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

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

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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

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

[19] T. Neumann and M. J. Freitag. Umbra: A disk-based system with in-memory performance. In *CIDR*, 2020.

[20] 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.

[21] 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.

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

[23] 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.

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

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