OptFSST: Optimized FSST String Compression

Hedi Chehaidar
Technical University of Munich
Germany
hedi.chehaidar@tum.de

Mihail Stoian
University of Technology Nuremberg
Germany
mihail.stoian@utn.de

Moritz Stargalla
University of Technology Nuremberg
Germany
moritz.stargalla@utn.de

Andreas Kipf
University of Technology Nuremberg
Germany
andreas.kipf@utn.de

ABSTRACT

Strings account for a substantial fraction of data in modern analytical systems, making lightweight compression with fast random access an important building block for efficient query processing. Fast Static Symbol Table (FSST) addresses this need by replacing frequent byte sequences with compact codes while preserving independent decompression of individual strings. However, FSST's compression effectiveness is limited by its greedy symbol selection and greedy encoding strategy, leaving encoding gains on the table.

We present OptFSST, an optimized FSST variant that improves its compression factors while preserving its static-symbol-table design and random-access decompression. OptFSST optimally encodes the text using dynamic programming given a symbol table. Additionally, we show that a generalized version of the symbol-table selection problem is NP-hard when the alphabet is part of the input, motivating heuristic table construction for field-level compressors. Hence, we add in OptFSST (i) an additional frequency counter that accelerates the discovery of longer symbols and (ii) a pruning strategy that removes redundant and conflicting symbol candidates during table construction. We also extend the same techniques to FSST12, yielding OptFSST12.

Our evaluation on 92 real-world string datasets shows that OptFSST improves the compression factors of FSST and FSST12 by up to 47.7% and 91.5%, with an average improvement of 7.3% and 17.0%, respectively, while retaining the fine-grained random-access properties. Notably, OptFSST12 improves FSST12's decompression speed by 1.2× on average.

VLDB Workshop Reference Format:
VLDB 2026 Workshop: 17th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS26).

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Hedi-Chehaidar/optfsst.

1 INTRODUCTION

Strings in Analytical Workloads. Strings are a central data type in analytical data management systems. They occur as URLs, identifiers, file paths, log messages, categorical attributes, free-form text, and semi-structured values originating from web, cloud, and enterprise applications. Studies of production analytical workloads and classic column-store systems show that strings can represent a substantial part of stored data and can dominate memory consumption in compressed columnar layouts [1, 9, 30]. Consequently, effective string compression is important not only for reducing storage size, but also for improving cache behavior, lowering memory-bandwidth pressure, and enabling efficient query processing on compressed data [21, 25, 37].

This memory-side pressure makes compact in-memory string representations valuable even when obtaining them requires additional compression-side work, provided that decompression remains fast and fine-grained.

Block-based Compression. General-purpose compressors such as Zstandard [14] and LZ4 [12] are highly effective on large contiguous byte streams, but do not support efficient access to individual strings. This motivates string compressors that preserve fine-grained random access while still reducing memory footprint.

FSST. Fast Static Symbol Table (FSST) was designed for this setting [8]. It replaces frequent byte sequences with compact codes from a static symbol table, allowing individual strings to be decompressed independently. This combination of compact strings, fast decompression, and random access makes FSST attractive for columnar storage formats and database systems. FSST-inspired dictionary compression is used or explored in systems and formats such as BtrBlocks [24], FastLanes [2], F3 [35], Lance [28], Vortex [32], and DuckDB's `DICT_FSST` [10] compression.

Limitations. The compression effectiveness of FSST depends on the quality of its symbol table and on the encoding decisions made with that table. The original FSST construction relies mainly on heuristics: it repeatedly compresses a sample, counts symbol and pair frequencies, and selects candidates according to local gain estimates. The final encoder then greedily emits the longest matching symbol at the current input position. These choices are fast, but they can be suboptimal. A locally frequent short symbol may block

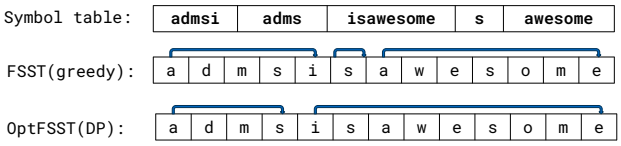


Figure 1: Example where FSST's greedy longest-match encoding is suboptimal. OptFSST yields the optimal compressed size given the symbol table.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

a better sequence of symbols, overlapping candidates can overestimate their true gain, and redundant candidates can occupy entries in the limited symbol table.

Example. Figure 1 gives a small example of the limitation caused by a greedy encoding. FSST selects the longest matching symbol at each position, even if this local decision prevents a better global segmentation of the remaining string. In the example, the symbol "`admsi`" is attractive locally, but choosing it leaves the remainder to be encoded using more symbols, resulting in 3 symbols in total. OptFSST finds the optimal strategy so as to minimize the compressed size, in this case 2 symbols.

OptFSST & OptFSST12. We present OptFSST, an improved FSST variant that targets these limitations while preserving FSST's byte-oriented format, static-symbol-table design, random-access property, and fast decompression path with a clear improvement in decoding speed in the case of OptFSST12. OptFSST improves both encoding and symbol-table construction: it uses dynamic programming to choose globally better encodings for a fixed symbol table, and it refines table construction with longer-symbol frequency counting and conflict-aware pruning. We also apply the same ideas to FSST12, yielding OptFSST12.

Contributions. In summary, we make the following contributions:

1. We introduce OptFSST, an optimized FSST variant that computes an optimal encoding for a given symbol table using dynamic programming while preserving FSST's decoder path and random-access property.
2. We show that selecting the optimal symbol table is NP-hard,[1] motivating heuristic table construction, and introduce longer-symbol frequency counting and conflict-aware pruning as a lightweight correction that improves many columns and reduces redundant symbol-table entries.
3. We extend the same techniques to FSST12 and evaluate OptFSST and OptFSST12 on 92 real-world string datasets, improving compression factors by up to 47.7% and 91.5%, with average improvements of 7.3% and 17.0%, respectively. OptFSST12 improves the average decompression speed by 1.2× over FSST12.

Paper Organization. The remainder of this paper is organized as follows. In §2, we discuss related work on string compression, block-based compression, dictionary compression, and FSST extensions. Then, in §3, we present the OptFSST design and implementation which combines dynamic-programming-based encoding, longer-symbol frequency counting, and conflict-aware symbol pruning. Notably, to motivate the introduced heuristics, we show in §4 that finding the optimal symbol table is indeed NP-hard. We evaluate in §5 OptFSST and OptFSST12 on real-world string datasets and quantify the trade-off between compression factor and compression speed, and conclude in §6.

2 RELATED WORK

Compression in Analytical Systems. Compression is a core component of modern analytical database systems. Besides reducing storage footprint, compressed representations improve cache locality, reduce memory bandwidth pressure, and can enable query execution directly on compressed data [21, 25, 29, 37]. Columnar systems in particular benefit from lightweight compression because values of the same type are stored contiguously and often exhibit repeated structure [1]. Much of the classic work on database compression focuses on numeric columns, dictionary encoding, run-length encoding, bit-packing, and other integer-oriented schemes. String data, however, remains more challenging: strings are variable-length, often have high cardinality, and may contain repeated substrings even when full values are unique.

Block-Based Compression. General-purpose compressors such as LZ4 [12], Zstandard [14], and Snappy [17] are widely used in storage and data processing systems because they provide strong throughput-compression trade-offs on large byte streams. All of the mentioned compressors are based on the Lempel-Ziv family of compression techniques [36]. LZ4 prioritizes speed and uses a simple block-oriented format with greedy match finding, while Zstandard combines LZ-style matching with entropy coding and a tunable compression-speed trade-off. Snappy, originally released by Google in 2011, emphasizes very fast compression and decompression with moderate compression ratios, making it a common choice for latency-sensitive storage and data-processing workloads.

These compressors are highly effective for sequential access and large blocks, but their block-oriented nature conflicts with fine-grained random access. Accessing one string typically requires decompressing the block that contains it, which can be wasteful for scans with selective predicates, joins, aggregations, dictionary lookups, or late materialization. Reducing block sizes improves access granularity, but usually hurts compression effectiveness.

Dictionary Compression for Strings. Dictionary encoding is the standard approach for compressing low-cardinality string columns. It stores each distinct string once and replaces column values with integer identifiers. This is highly effective when many values repeat exactly, and it enables efficient query processing over integer codes. However, dictionary encoding does not compress the dictionary strings themselves. For high-cardinality string columns, such as URLs, identifiers, paths, or log messages, many strings are similar but not identical, which limits the effectiveness of pure dictionary encoding. `DICT_FSST` addresses this limitation by combining dictionary encoding with FSST: the column is first deduplicated into a dictionary of unique strings, and FSST is then applied to compress the dictionary payload itself [10]. This hybrid design exploits both full-string repetition and repeated substrings inside unique dictionary entries, making it a natural integration point for improvements to the underlying FSST compressor.

FSST. Fast Static Symbol Table (FSST) was designed specifically for database string columns that require fast decompression and random access [8]. FSST builds a static symbol table from a sample of the input and replaces frequent byte sequences of length up to eight with one-byte codes. Bytes that are not represented by a symbol-table entry are emitted as escaped characters: FSST writes a reserved escape code (255) followed by the original byte, so such characters cost two output bytes.

Since the symbol table is immutable during decompression, each compressed string can be decoded independently. This design allows compressed strings to remain byte strings, simplifying system integration and enabling equality comparisons on compressed data.

[1] When the alphabet is part of the input; we refer the reader to §4 for more details.

The main algorithmic challenge in FSST is constructing a good symbol table. The original algorithm uses an iterative bottom-up procedure: it compresses a sample with the current table, counts symbol and pair frequencies, and constructs the next generation of candidate symbols from high-gain symbols and their concatenations [8]. This process addresses some dependencies between overlapping symbols by observing which symbols are actually used during compression. However, FSST still relies on greedy decisions. During encoding, it selects the longest matching symbol at the current position. During table construction, it ranks candidates using local gain estimates. The FSST paper already identifies symbol dependencies and overlapping candidates as central difficulties in symbol-table construction [8]. OptFSST builds directly on this observation by replacing greedy encoding decisions with a dynamic-programming formulation and by improving candidate selection through additional frequency counting and pruning.

FSST Extensions. FSST+ is a recent extension that targets a different source of redundancy: common prefixes across neighboring strings [4]. FSST+ groups strings into small blocks, sorts them locally, and uses dynamic programming to select prefix-sharing groups. Shared prefixes are stored once and referenced by suffixes, while the prefix and suffix data are still compressed using FSST. This makes FSST+ complementary to OptFSST. FSST+ improves the layout around FSST by exploiting cross-string prefix redundancy, whereas OptFSST improves the internal FSST symbol table and encoding decisions. As a result, OptFSST can potentially benefit systems that already use FSST as a component, including `DICT_FSST` and prefix-oriented extensions such as FSST+.

GPU-aware FSST. Recent GPU-enabled query engines have renewed interest in making column encodings and compression schemes accelerator-friendly [19, 22, 34]. In parallel, recent work has shown that GPU-aware encodings and file-format configurations can substantially improve analytical scan performance [3, 20, 26], in particular for string data via a GPU-aware FSST adaptation [5, 31]. OptFSST's DP can also be GPU-accelerated, particularly via NVIDIA Blackwell's recent DPX instructions [27].

OnPair. OnPair is a recent field-level string compressor for in-memory workloads that, like FSST, compresses strings independently and therefore supports fine-grained random access [15]. In contrast to FSST's small static symbol table with one-byte codes and symbols of length up to eight bytes, OnPair uses a larger dictionary with fixed two-byte token identifiers and builds the dictionary by incrementally merging frequent adjacent token pairs during a sequential pass over a sample. Its OnPair16 variant bounds dictionary entries to 16 bytes to make longest-prefix matching and decompression more hardware friendly. Thus, OnPair explores a different point in the design space: it utilizes pair merging and larger dictionaries to improve compression effectiveness, whereas FSST prioritizes a compact fixed-size symbol-table, faster encoding, and simple one-byte symbols.

Positioning of OptFSST. OptFSST differs from block-based compressors by preserving FSST's random-access decompression model. It differs from dictionary and front-coding approaches by compressing substrings inside individual string values rather than relying on exact duplicates or lexicographic adjacency. It also differs from FSST+ by optimizing FSST's symbol selection and encoding process instead of introducing a new prefix-sharing layout. The closest baseline is therefore the original FSST algorithm. OptFSST keeps the same high-level abstraction: a static table of byte-sequence symbols, but replaces parts of the greedy construction and encoding pipeline with globally informed decisions.

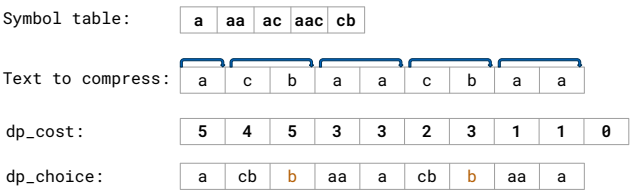


Figure 2: Dynamic-programming-based encoding example. OptFSST computes the optimal suffix cost for each input position and follows the selected decisions to obtain a globally better encoding (marked by the arrows) than greedy longest-match selection. Escaped characters are marked in brown.

As a footnote in FSST's original paper [8], the authors do indeed provide a DP approach as an alternative, but without reporting any performance numbers. This motivated our work, in which we extensively benchmark this approach, optimize the "severely affected" encoding, i.e., the inner DP loop, and extend the analysis to FSST12, which got adopted in CWI's FastLanes [2, Table 1].

3 APPROACH

OptFSST improves FSST by modifying only the compression and symbol-table construction pipeline. The decompression format remains unchanged: compressed strings are still byte sequences of symbol codes and escape codes interpreted through a static symbol table. As a result, OptFSST preserves FSST's random-access decompression property while improving compression factors through three contributions: dynamic-programming-based encoding, a third frequency counter, and symbol pruning. We also apply the same ideas to FSST12, resulting in OptFSST12.

3.1 Dynamic-Programming-Based Encoding

Motivation. As discussed above, FSST's longest-match encoder is fast but not optimal for a fixed symbol table. Since each symbol code costs one byte and an escaped character costs two bytes, fixed-table encoding can be formulated as finding the minimum-cost segmentation of the input string. OptFSST solves this problem using dynamic programming.

Dynamic Programming Formulation. For an input string s of length n, OptFSST defines $dp[i]$ as the minimum compressed size of the suffix $s[i:n]$. The base case is $dp[n] = 0$. At each position i, the encoder can either escape the current byte, which costs two output bytes, or emit one symbol from the current symbol table T

```
1  void BuildDP(const uint8_t* data, int n,
2               const vector<TrieNode>& trie,
3               vector<uint32_t>& dp_cost,
4               vector<uint16_t>& dp_choice) {
5    dp_cost.assign(n + 1, 0);
6    dp_choice.assign(n, 255);  // Default: escape.
7
8    dp_cost[n] = 0;  // Base case.
9
10   for (int i = n - 1; i >= 0; --i) {
11     uint32_t best_cost = 2 + dp_cost[i + 1];
12     uint16_t best_code = 255;
13
14     int node = 0;
15     uint32_t limit = min(Symbol::maxLength, (uint32_t)(n - i));
16
17     for (uint32_t off = 0; off < limit; ++off) {
18       uint8_t byte = data[i + off];
19       node = trie[node].child[byte];
20
21       if (node == -1)
22         break;
23
24       int code = trie[node].symbol_code;
25       if (code != -1) {
26         uint32_t len = off + 1;
27         uint32_t cost = 1 + dp_cost[i + len];
28
29         if (cost <= best_cost) {
30           best_cost = cost;
31           best_code = code;
32         }
33       }
34     }
35
36     dp_cost[i] = best_cost;
37     dp_choice[i] = best_code;
38   }
39 }
```

Listing 1: Core dynamic-programming construction used by OptFSST.

that starts at i, which costs one output byte. The recurrence is:

$$dp[i] = \min\left(2 + dp[i+1], 1 + \min_{\substack{i<j\le n \\ s[i:j]\in T}} dp[j]\right).$$

In addition to the cost array, OptFSST stores an array `dp_choice[i]` containing one best decision for position i. After filling both arrays from right to left, compression follows the decisions stored in `dp_choice`, as shown graphically in Figure 2.

Trie-based Lookup. A direct implementation of the recurrence would test all symbols at every input position. OptFSST avoids this by storing the current symbol table in a trie. Starting from position i, the encoder walks the trie byte by byte for at most the maximum FSST symbol length. If the traversal reaches a terminal node, the represented symbol matches the current input prefix. If no child exists for the next byte, traversal stops immediately. This avoids full-symbol comparisons and limits candidate enumeration to symbols that actually match the input.

Implementation. Listing 1 shows the core of the DP implementation. Each position is initialized with the escape cost. Then OptFSST traverses the trie of symbols starting at the current byte position. Whenever a terminal trie node is reached, the corresponding symbol is a valid candidate and can update the best cost. Experiments showed that preferring longer symbols, as a tie-breaker for equal DP costs, yields better overall compression factors; hence the use of "≤" in the cost update condition in line 29. The final implementation computes the dynamic-programming cost and the selected code in the same pass.

Use During Table Construction. OptFSST uses the DP encoder both for final compression and during symbol-table construction. FSST builds its symbol table iteratively: in each generation, it compresses a sample using the current table, counts emitted symbols and pairs, and then constructs the next generation from high-gain candidates. Replacing the greedy encoding in this phase changes the frequency signal used for table construction. Counts now reflect symbols that participate in globally good encodings rather than symbols selected only because they are locally longest, improving the feedback loop between compression and symbol selection.

Traversal Optimizations. The dynamic-programming step introduces additional compression work, with trie traversal being the main cost. Therefore, OptFSST optimizes the `BuildDP` traversal path using loop unrolling, compiler branch-prediction hints, and small implementation-level refinements. These optimizations do not change the recurrence or the selected encoding; they merely reduce the cost of computing the `dp_cost` and `dp_choice` arrays. We evaluate the effect of these optimizations in §5.3.

Adaptive and Vectorized Variants. We also evaluated two variants aimed at reducing the DP overhead. First, we tested an adaptive strategy that switches to greedy full-text encoding only when a DP-based compression pass over the training sample exceeds a configurable compressed-size threshold. This did not contribute positively to the compression-factor to compression-speed tradeoff compared to always choosing the DP path. Second, we tested a SIMD-oriented bucket layout that groups symbols by their first bytes and vectorizes candidate comparisons within the selected bucket. In our experiments, this layout was slower than the optimized trie traversal, because bucket lookup and candidate filtering dominated the benefit of vectorized comparisons. We therefore retain the trie-based DP implementation.

3.2 Faster Convergence

Motivation. FSST's original construction counts individual symbols through an array `count1`, and adjacent symbol pairs with a 2D array `count2`. Pair counts allow the next generation to form longer candidates by concatenating two symbols that occur next to each other. However, longer recurring patterns may require multiple generations before they become visible as candidates. This slows convergence and can prevent useful long symbols from entering the table early enough.

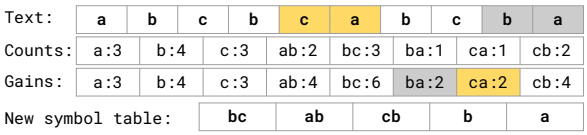


Figure 3: Example of symbol selection without pruning. The symbols "ba" and "ca" are not chosen, because symbols "a" and "b" had higher static gains. OptFSST handles this case optimally.

Counting Triples. OptFSST introduces a third frequency counter, `count3`, for triples of consecutive emitted symbols.[2] During sample compression, OptFSST records patterns of the form (a, b, c) in addition to the original single-symbol and pair counts. Candidate generation can then concatenate three adjacent symbols directly, as long as the resulting symbol respects the maximum symbol length. This exposes longer recurring patterns earlier than pairwise growth alone. We also experimented with extending this idea to higher-order counters, beginning from `count4`, but they did not yield significant additional improvements in our experiments.

Compact Representation. A dense three-dimensional counter would be too large. Since only triples that actually occur in the compressed sample are relevant, OptFSST stores `count3` sparsely in a hash map. Each FSST code fits into nine bits, so three codes can be packed into a 32-bit integer.

Candidate Generation. During the new symbol table population of each generation, OptFSST considers the original FSST candidates from `count1` and `count2`, and additionally iterates over the non-zero entries of `count3`. For every observed triple, the three corresponding symbols are concatenated and inserted into the candidate heap if the resulting symbol is short enough. The gain is computed in the same style as FSST, multiplying the candidate length by the observed frequency. Thus, OptFSST preserves FSST's generation-based construction, but extends the candidate space with longer combinations discovered by the third counter.

3.3 Symbol Pruning

Motivation. FSST ranks candidate symbols by static gain, which is based on frequency and symbol length. This estimate can be inaccurate when symbols overlap. If a larger symbol is selected, smaller symbols contained in it may no longer be emitted often in the next generation. Without correction, these smaller symbols can still appear valuable and occupy entries in the limited symbol table. OptFSST introduces pruning to partially correct this overestimation during candidate selection.

Example w/o Pruning. Figure 3 illustrates the issue for a table with capacity five. Static gain selects the high-count one-byte symbols "a" and "b", although replacing them with the two-byte symbols "ba" and "ca" would cover the text more effectively. The example shows that gains computed before candidate selection can become stale once overlapping longer symbols enter the table.

Pruning Logic. When OptFSST selects a concatenated symbol, it reduces the counts of the smaller symbols and sub-combinations used to form it. For example, if a selected candidate corresponds to the concatenation XYZ, the counts of X, Y, Z, XY, and YZ are reduced. Their gains are then recomputed before they can be selected. Candidates whose corrected gain becomes non-positive are discarded. This gives other candidates the opportunity to enter the table and reduces redundancy among selected symbols.

Returning to the example of Figure 3, OptFSST would reduce the counts of the symbols "b" and "c" by the count of "bc", which is 3, right after "bc" is added to the new symbol table. In this way, the symbol "c" will be immediately discarded as its count reached 0, but the symbol "b" still remains with the count 1 and gets reinserted into the candidate heap. The symbol "b" will be discarded after adding "ab" and the symbol "a" will be discarded after adding the fourth symbol. Then we would have all 2-byte symbols in the new symbol table, which would compress the given text optimally.

Implementation. Listing 2 shows a simplified version of the pruning step. Candidate gains stored in the heap may become stale after earlier pruning operations, so every popped candidate is validated against its current corrected count after being reconstructed from its individual parts through concatenation. If the candidate is still valid, it is inserted into the next symbol table and its parts are pruned.

3.4 OptFSST12

We also apply the same three contributions to FSST12, resulting in OptFSST12. FSST12 provides a larger symbol space and uses a different symbol storage design, but the main optimization problem remains the same: the compressor must decide which symbols to place in the table and how to encode strings with them. OptFSST12 therefore uses the dynamic-programming encoder, the third frequency counter, and pruning in the FSST12 setting.

Preserved Properties. Both OptFSST and OptFSST12 preserve the central properties of FSST. The symbol table remains static during decompression, each compressed string can still be decompressed independently, and the decompression path is unchanged. The additional computation is concentrated in compression and table construction, where more informed optimization can improve compression factors. This design deliberately trades additional compression effort for better compressed size while keeping FSST's fast random-access decompression model intact.

4 OPTIMAL SYMBOL TABLE SELECTION

FSST uses a greedy strategy in both the symbol-table construction and encoding phase. While OptFSST guarantees the optimal encoding with respect to a *given* table, finding the symbol table is still suboptimal (even though we are using DP as a subroutine to evaluate the quality of the current table). A natural question is whether the problem is inherently hard, thereby justifying the proposed heuristics, or whether we simply have not yet found an

[2]The motivation is that a Markov model, as (implicitly) used in the pair counting of FSST (`count2`), can better capture the distribution when conditioned on a longer context; in this case, of length two.

```
struct Candidate {
  uint16_t a;
  uint16_t b;
  uint16_t c;
  uint8_t arity;  // 1, 2, or 3.
  uint32_t gain;
};

while (next_table.size() < max_symbols && !heap.empty()) {
  Candidate candidate = heap.PopMax();

  Symbol symbol = Reconstruct(candidate);
  uint32_t current_count = CorrectedCount(candidate);
  uint32_t current_gain = current_count * symbol.length();

  // Ignore stale heap entries.
  if (current_gain != candidate.gain || current_gain == 0)
    continue;

  if (next_table.Contains(symbol))
    continue;

  next_table.Insert(symbol);

  // Correct counts of covered smaller candidates.
  if (candidate.arity >= 2) {
    PruneSymbol(candidate.a, current_count);
    PruneSymbol(candidate.b, current_count);
  }

  if (candidate.arity == 3) {
    PruneSymbol(candidate.c, current_count);
    PrunePair(candidate.a, candidate.b, current_count);
    PrunePair(candidate.b, candidate.c, current_count);
  }
}
```

Listing 2: Simplified candidate selection with pruning.

efficient algorithm. In this work, we show the problem of selecting an optimal symbol table is indeed NP-hard.[3]

We formalize the problem in the following, emulating the setting we have in FSST:

Problem 1 (FSST Symbol Table Selection)**.** *Fix an integer $\ell \geq 1$. An instance of STS$_\ell$ consists of:*

- *a finite multiset $\mathcal{T}$ of strings over an alphabet Σ,*
- *an integer $K \geq 0$, and*
- *a target bound $B \geq 0$.*

A dictionary D is a set of strings over Σ of length at most ℓ, with $|D| \leq K$. A D-encoding *of a string $s \in \mathcal{T}$ is a partition of s into consecutive phrases, each of which is either:*

- *a literal byte $c \in \Sigma$, encoded at cost 2 (escape byte plus c), or*
- *an occurrence of a dictionary symbol $w \in D$, encoded at cost 1.*

The cost of an encoding is the sum of phrase costs, and $\mathrm{cost}_D(s)$ denotes the minimum cost of any D-parsing of s. Finally,

$$\mathrm{cost}_D(\mathcal{T}) \;:=\; \sum_{s \in \mathcal{T}} \mathrm{cost}_D(s).$$

[3]Our reduction uses an alphabet whose size grows with the input. If both ℓ and the alphabet size are fixed, then the problem is solvable in polynomial time; see Appendix B. If instead the dictionary size is fixed, the problem is also solvable in polynomial time; compare [8, Sec. 4.1].

The decision problem asks whether there exists a dictionary D with $|D| \leq K$ such that

$$\mathrm{cost}_D(\mathcal{T}) \leq B.$$

We show that STS$_\ell$, the problem of selecting symbols of length at most ℓ (including escaping single bytes), is NP-hard:

Theorem 1. *STS$_\ell$ is NP-hard for every fixed $\ell \geq 2$.*

We refer the reader to Appendix A for the proof of Theorem 1. Note that, unlike the tokenization problem found in modern language models, the STS$_\ell$ problem fixes the length ℓ of a token to be considered; in FSST, a symbol's length is bounded, e.g., 8 or 12. Hence, the recent NP-hardness proofs for the tokenization problem, as in Ref. [23, 33], are not directly applicable. Our proof thus contributes to the hardness landscape for the tokenization problem.

5 EVALUATION

This section evaluates OptFSST and OptFSST12 with respect to compression-factor improvements and the effect on compression and decompression speed. We first describe the benchmark datasets and methodology. We then present an ablation study that isolates the contribution of each optimization. Finally, we analyze the runtime impact of the proposed changes.

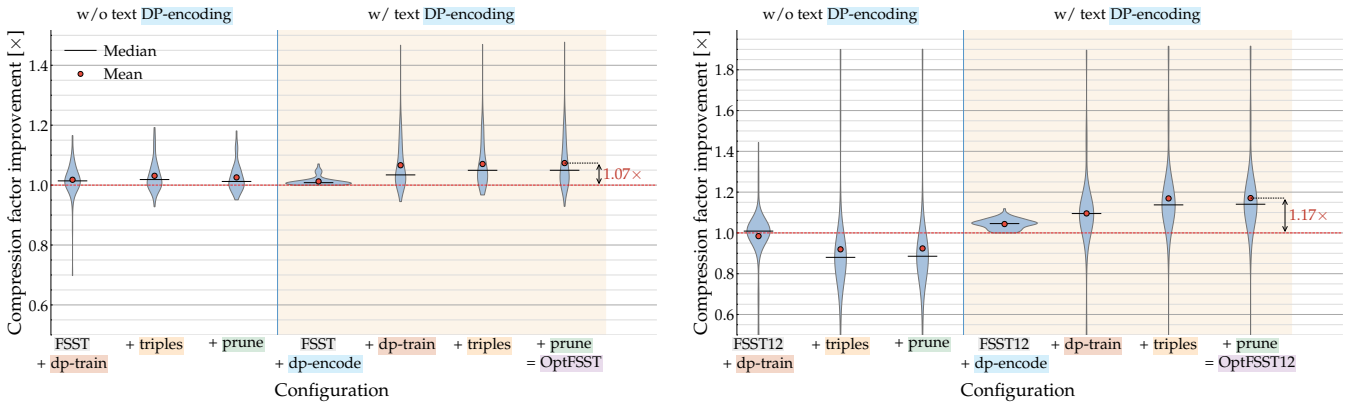
Setup. We conduct the experiments on a single node Intel® Xeon® Gold 5318Y CPU (24 cores, 48 hyper-threads). The machine is equipped with 128GB DDR4 main memory and runs Ubuntu 24.04. All experiments are run single-threaded.

5.1 Benchmark Datasets

We evaluate OptFSST on heterogeneous real-world string datasets. The goal is to cover different classes of string columns, including natural language text, URLs, identifiers, semi-structured strings, and short categorical strings. We use the following datasets:

- **dbtext** (23 files): the database string corpus introduced with FSST, containing representative string columns from database workloads [7].
- **NextiaJD** (22 files): a collection of real-world datasets used for data discovery experiments, containing heterogeneous string columns such as URLs, comments, and structured identifiers [13].
- **Public BI Benchmark** (31 files): a business-intelligence benchmark containing realistic analytical columns, including textual and categorical attributes [16].
- **CyclicJoinBench** (13 files): a benchmark for join-heavy analytical workloads that also contains string-typed columns [18].
- **ClickBench** (3 files): a clickstream analytics benchmark with web-oriented string columns such as URLs, referrers, user-agent strings, titles, and categorical attributes [11].

We focus on string columns for which FSST is meaningful. Columns that are compressed better using dictionary encoding are filtered out. We also require a sufficiently large number of strings per column (at least 1000) to obtain stable compression and runtime measurements. Each configuration is evaluated on the same set of columns and compared against the corresponding baseline, FSST or FSST12.



(a) OptFSST **(b) OptFSST12**

Figure 4: Compression-factor improvement over the corresponding baseline. The dashed red line marks parity with FSST or FSST12. Each box shows the distribution across benchmark columns, and the red point denotes the mean.

We report compression-factor improvement as a multiplicative ratio over the baseline:

$$\text{improvement} = \frac{\text{compression factor of variant}}{\text{compression factor of baseline}}.$$

Values above 1.0 therefore indicate an improvement over the baseline, while values below 1.0 indicate a regression. Runtime results are reported in MB/s. For compression speed, we measure only the final corpus compression after the symbol table has already been constructed. Table construction speed is measured separately, since the FSST SIMD version does not affect this phase.

5.2 Compression Factor Improvement

OptFSST. Figure 4a shows the ablation study for OptFSST. The first configuration, *FSST + dp-train*, uses dynamic programming during symbol-table construction. This already improves compression factors by 1.02× on average. The effect is moderate because only the training signal changes: the final corpus is still encoded with the baseline greedy encoder. The *+ triples* configuration adds the third frequency counter and reaches an average improvement of 1.03×. This shows that exposing longer recurring symbol combinations earlier during table construction helps the algorithm discover more useful symbols.

The *+ prune* configuration adds pruning during candidate selection. In terms of the aggregate mean, pruning has no noticeable effect on the final compression factor. However, the aggregate mean hides the fact that pruning is beneficial for a majority of columns. For OptFSST, pruning improves the final compression factor for 56.5% of the files, with the best case reaching a 1.05× improvement for the sparse column `HashTags_1::null_21` from the `Public BI Benchmark` dataset. Without text DP encoding, individual columns still benefit substantially, with the best case reaching 1.113×. Thus, pruning is not the main source of the headline average improvement, but it consistently helps in the cases where overlapping candidates would otherwise lead to stale or overestimated gains.

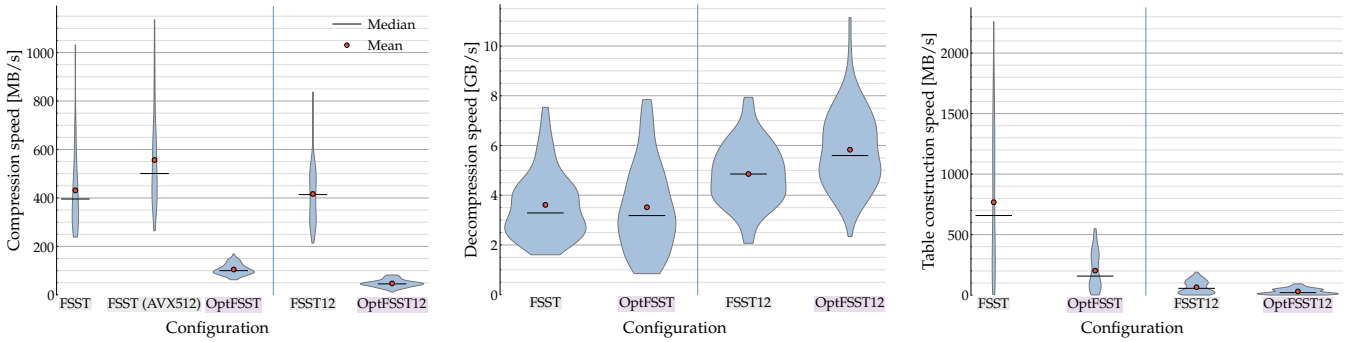
The *FSST + dp-encode* configuration applies dynamic programming to the final corpus encoding and reaches 1.01× on average. This confirms that even for a fixed symbol table, greedy longest-match encoding can leave compression opportunities unused.

The final OptFSST configuration combines DP training, triple counting, dynamic-programming encoding, and pruning. It reaches an average compression-factor improvement of 7.3% over FSST. OptFSST improves the compression factor over FSST on 82/92 benchmark columns (89%). The main aggregate gain comes from the two dynamic-programming uses and from the additional triple counter.

The violin plot also shows that the improvement is not uniform across all columns. Some columns remain close to parity, while others benefit substantially more, with outliers reaching up to 47.7% improvement, which is observed in a column with short strings of length 2. This variation is expected because OptFSST mainly improves cases where symbol overlap and greedy encoding decisions matter. Columns that are already well represented by FSST's greedy table construction leave less room for improvement.

OptFSST12. Figure 4b repeats the ablation study for FSST12. The results show that the same ideas remain effective even when the baseline has a larger symbol space. However, the individual contributions behave differently. The *FSST12 + dp-train* configuration is slightly below parity on average, at 0.98×. This indicates that changing only the training encoder can sometimes disturb FSST12's symbol selection when the final encoding remains unchanged. In contrast, the *+ triples* configuration improves the average to 1.04×, showing that longer candidate discovery is still useful in the larger-symbol-space setting.

For FSST12, pruning again has no noticeable aggregate effect on the final compression factor. As with OptFSST, this neutral mean should not be interpreted as pruning being unused. Pruning improves the final compression factor for 54.3% of the files, and the best final-encoding case reaches 1.03×. Without text DP encoding, the effect is even more visible for FSST12: pruning improves the compression factor for 67.4% of the files, with a best case of 1.19× on the column `c_name` from the `dbtext` dataset. This suggests that the larger symbol space creates more opportunities for overlapping



(a) Final corpus compression speed [MB/s] **(b) Decompression speed [GB/s]** **(c) Symbol-table construction speed [MB/s]**

Figure 5: Runtime comparison. Compression speed measures only final corpus compression after the symbol table has been constructed. Table construction is reported separately because OptFSST performs additional optimization during this phase.

Table 1: Runtime ablation. Values are arithmetic-mean slowdowns (×) over the corresponding FSST/FSST12 baseline. E2E includes both table construction and corpus encoding.

	OptFSST		OptFSST12	
Configuration	Symbol table	E2E	Symbol table	E2E
Baseline	1.00	1.00	1.00	1.00
+ DP train	2.49	1.68	1.51	1.45
+ triples	4.02	2.36	2.22	2.07
+ prune	4.07	2.38	2.28	2.12
Baseline + DP encode	n/a	2.33	n/a	1.84
All configurations	4.07	3.82	2.16	3.02

candidates to compete, making stale gain correction useful even when the final aggregate mean remains unchanged. We therefore keep pruning in the final configuration as a lightweight correction for stale gain estimates caused by overlapping candidates, rather than as a mechanism that is expected to increase the aggregate mean on every workload.

The final OptFSST12 configuration combines all previously mentioned contributions and achieves an average compression-factor improvement of 1.17× over FSST12. OptFSST12 improves over FSST12 on 87/92 columns (95%). These numbers are larger than the relative improvement observed for OptFSST over FSST, suggesting that FSST12's larger symbol space creates more opportunities for improved candidate selection.

Overall, the ablation study shows that dynamic programming and triple counting are the main sources of compression-factor improvement. Dynamic-programming encoding improves the final segmentation decisions, while the third counter exposes longer recurring candidates earlier during table construction. Pruning serves as a correction mechanism that proved useful for most files.

5.3 Effect on (De)Compression Speed

Runtime Ablation. Table 1 complements the compression-factor ablation in §5.2 by separating table-construction overhead from end-to-end compression time; we report the slowdown over the corresponding baseline. Note that the rows are cumulative for the training-side variants, i.e., + *triples* includes DP-based training, and + *prune* includes both DP-based training and triple counting. In the case of full-text DP encoding only, the table-construction time remains unchanged. We observe that, for table construction, DP-based training and triple counting are the main sources of overhead, while pruning has no noticeable effect. The relative construction overhead is smaller for FSST12 because its baseline table construction is already more expensive due to the larger symbol space.

OptFSST. Next, we analyze the throughput of OptFSST's compression, decompression, and symbol-table construction.

Compression. Figure 5a shows that OptFSST reduces final corpus compression speed compared with FSST. FSST compresses at 431.9 MB/s on average, while OptFSST reaches 104.9 MB/s.[4] The FSST SIMD version is around 1.3× faster than the regular FSST, which increases the gap. This slowdown is expected because OptFSST replaces the branchless greedy encoder with a dynamic programming encoder that evaluates multiple candidate symbols per input position. Notably, the optimized trie traversal, including loop unrolling and compiler branch-prediction hints, reduces this overhead, as shown in Figure 6: the corpus encoding speed is improved by these optimizations by 1.03× and 1.06× on average in OptFSST and OptFSST12, respectively. This helps further close the gap to FSST's compression performance.

Decompression. Figure 5b shows that decompression is essentially unaffected. FSST reaches 3.6 GB/s on average, while OptFSST reaches 3.5 GB/s. Since OptFSST does not change the compressed representation or the decompression algorithm, decompression remains a simple symbol-table lookup per emitted code. Small differences in the measured decompression speed are therefore mainly caused by memory behavior and branch prediction, as the placement of escape codes in the compressed text affects the behavior of the decompression function in FSST.

[4]Note that a "natural" slowdown would be 8x, as the inner DP loop has to run on all 8 possible lengths. However, the trie, as explained in §3.1, allows exiting non-matching paths faster on average.

Table 2: Comparison of FSST-style and block-based compressors.

Metric	Compression factor [×]				Compression speed [MB/s]				Decompression speed [GB/s]			
	min	mean	median	max	min	mean	median	max	min	mean	median	max
FSST	1.17	2.68	2.33	6.54	238.36	431.93	395.55	1032.01	1.55	3.60	3.28	7.22
FSST (AVX512)	1.17	2.68	2.33	6.54	264.66	556.56	500.69	1135.60	1.55	3.60	3.28	7.22
OptFSST	1.21	2.93	2.48	7.25	61.28	104.93	99.30	166.78	0.85	3.54	3.18	7.96
FSST12	0.97	2.29	2.31	4.46	213.84	416.77	409.77	849.51	1.99	4.91	4.89	8.40
OptFSST12	0.39	2.72	2.60	5.13	15.23	47.48	45.59	82.22	2.55	5.86	5.87	9.16
Snappy	1.00	3.62	2.45	19.24	49.69	272.59	244.40	714.27	0.11	0.57	0.48	1.71
LZ4	1.00	5.81	2.36	87.63	76.33	448.88	335.26	1989.84	0.50	1.61	1.26	6.00
ZSTD	1.55	10.83	4.02	174.34	14.38	164.18	137.29	681.75	0.07	0.61	0.50	2.75

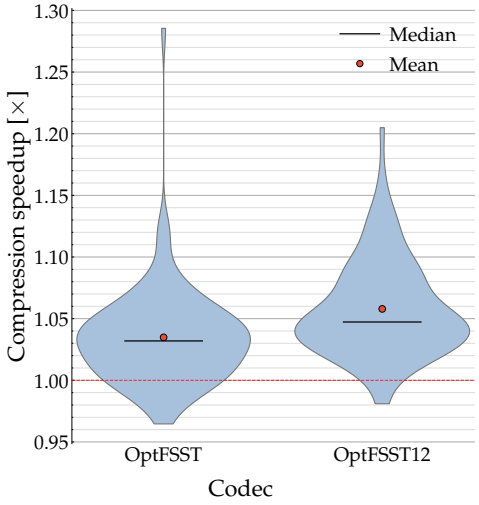


Figure 6: Speedup of corpus compression after the trie traversal optimizations—loop unrolling and branch-prediction hints—in both OptFSST and OptFSST12.

Table Construction. Figure 5c shows that during symbol-table construction, baseline FSST constructs tables at 766.8 MB/s on average. OptFSST constructs tables at 200.7 MB/s. As shown in Table 1, the decrease is caused mainly by dynamic-programming training and triple counting. Pruning has no noticeable effect on construction time, since it operates on the candidate heap after frequency counting. Nevertheless, table construction remains practical, especially because it is performed once per block, row group, or compression unit, while decompression is usually on the critical path for query execution.

OptFSST12. Table construction for FSST12 is substantially slower than FSST, reaching 64.4 MB/s on average in our measurements. OptFSST12 further reduces table construction speed to 31.6 MB/s. This overhead is expected because OptFSST12 applies the same additional optimization steps as OptFSST, but in the setting of a larger symbol space. The larger candidate space increases the cost of construction, and the dynamic-programming and pruning steps add further work.

For final compression speed, OptFSST12 follows the same trend as OptFSST: the dynamic-programming encoder is slower than the greedy baseline because it computes globally optimal suffix costs instead of emitting the first longest match. However, this cost is paid during compression, not decompression. Unlike OptFSST, OptFSST12 noticeably improved the average decompression speed of FSST12 by 1.2×. This is due to the higher improvement of the compression factor, and thus smaller compressed size, combined with the simpler decompression design of FSST12 that does not contain branching caused by escape codes like in FSST. Thus, OptFSST12 trades additional compression and construction effort for improved compression factors and decompression speed.

Trade-off and Hardware Implications. The runtime results quantify the intended trade-off: OptFSST and OptFSST12 spend additional CPU time during compression and table construction to produce a more compact representation that is later scanned, cached, and decoded. This is attractive in analytical systems because compression is typically paid during ingest, compaction, or row-group creation, whereas compressed columns may be read many times during query processing. On modern CPUs, analytical scans are often constrained by memory traffic and cache residency. Reducing the string payload can therefore improve the amount of useful data per cache line and lower memory-bandwidth pressure. The branch-heavy DP work is placed on the less frequent compression side, while the frequently executed decompression side continues to use the simple static-table decoding path.

5.4 Comparison with Block-Based Compressors

Compression Factor. Table 2 compares the FSST-based compressors with the block-based compressors Snappy, LZ4, and ZSTD. The block-based compressors achieve higher compression factors on average because they can exploit redundancy across larger byte ranges. Snappy reaches 3.62× on average, which is 1.2× higher than OptFSST, whereas LZ4 and ZSTD achieve much larger average compression factors, which are 1.9× and 3.6× higher than that of OptFSST, respectively. This illustrates the main design trade-off: block-based compressors obtain better compression by using a larger compression context, while FSST-style compressors sacrifice some compression effectiveness to preserve fine-grained random access and independent decompression of individual strings.

The minimum compression factor of OptFSST12, 0.39×, is an outlier on the column `place/name` from the `CyclicJoinBench (SNB1)` dataset. This column is small (13.6 KB) and consists almost entirely of distinct (only one duplicate) short strings (8.2 bytes on average). DP training fills the entire 4096-entry symbol table: Its serialization alone exceeds twice the input size and dominates the reported compression factor, while FSST12 constructs a smaller table on the same column. As a result, OptFSST12 can lose on this column even though its mean and median compression factors remain higher than FSST12.

Compression and Decompression Speed. The speed results in Table 2 show the opposite side of this trade-off. During compression, the AVX512 version of FSST is the fastest method on average, followed closely by LZ4 and FSST12. In contrast, OptFSST and OptFSST12 are slower due to their dynamic-programming encoder. Compared with LZ4, OptFSST is 4.2× slower and OptFSST12 is 9.4× slower on average. The improved compression factors of OptFSST(12) therefore come at a clear compression-time cost.

However, decompression shows why the FSST-based design remains attractive for analytical workloads. On average, OptFSST12 decompresses at 5.86 GB/s, which is 3.6× faster than LZ4, 10.2× faster than Snappy, and 9.6× faster than ZSTD.

Overall, the comparison shows that OptFSST and OptFSST12 occupy a different point in the design space than general-purpose block compressors. They do not compete with ZSTD or LZ4 on maximum compression factor, and they are slower during compression than the fastest baselines. Instead, they improve the compression effectiveness of FSST-style random-access string compression, thereby improving its high decompression throughput.

6 CONCLUSION & FUTURE WORK

This paper presented OptFSST, an optimized FSST variant that improves compression effectiveness while preserving FSST's static-table format, independent decompression, and random-access interface. The key idea is to spend additional work during compression and table construction, through DP-based encoding, triple counting, and pruning, while leaving the decompression-side abstraction unchanged. Our evaluation shows that this trade-off improves compression factors by 7.3% over FSST and 17.0% over FSST12 on average, with decompression preserved for OptFSST and improved for OptFSST12.

The ablation study confirms that the three contributions are complementary: dynamic-programming-based encoding improves final compression decisions, triple counting accelerates the discovery of longer useful symbols, and pruning reduces wasted symbol-table capacity.

Future Work. Future work should focus on closing the remaining compression-speed gap to FSST. The most immediate direction is to further optimize `BuildDP`, especially the trie traversal and candidate lookup logic, since this is the dominant added cost of dynamic-programming-based encoding. A second direction is to explore more adaptive variants of the proposed techniques, for example enabling pruning or triple counting only when the sampled data suggests that they are beneficial.

REFERENCES

[1] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. 2008. Column-Stores vs. Row-Stores: How Different Are They Really?. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. 967–980. https://doi.org/10.1145/1376616.1376712

[2] Amir Afroozeh and Peter Boncz. 2025. The FastLanes File Format. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4629–4643. https://doi.org/10.14778/3749646.3749718

[3] Azim Afroozeh, Lotte Felius, and Peter Boncz. 2024. Accelerating GPU Data Processing using FastLanes Compression. In *20th International Workshop on Data Management on New Hardware (DaMoN)*. ACM. https://doi.org/10.1145/3662010.3663450

[4] Yan Lanna Alexandre. 2025. *FSST+: Enhancing String Compression Through Common Prefix Extraction*. Master's thesis. Vrije Universiteit Amsterdam and Universiteit van Amsterdam, Amsterdam, The Netherlands.

[5] Tim Anema, Joost Hoozemans, Zaid Al-Ars, and H. Peter Hofstee. 2025. High Throughput GPU-Accelerated FSST String Compression. In *16th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*.

[6] Nicola Apollonio and Bruno Simeone. 2014. The maximum vertex coverage problem on bipartite graphs. *Discret. Appl. Math.* 165 (2014), 37–48. https://doi.org/10.1016/J.DAM.2013.05.015

[7] Peter Boncz. 2020. cwida/fsst. https://github.com/cwida/fsst. Original release: 2020-03-03. Accessed: 2026-01-10.

[8] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: Fast Random Access String Compression. *Proceedings of the VLDB Endowment* 13, 11 (2020), 2649–2661. https://doi.org/10.14778/3407790.3407851

[9] Peter A. Boncz and Martin L. Kersten. 1995. Monet: An Impressionist Sketch of an Advanced Database System. In *Proceedings of the IEEE BIWIT Workshop*. San Sebastian, Spain, 240–251.

[10] Tijmen Bruineman. 2025. [Compression] Introduce DICT_FSST Compression Method. https://github.com/duckdb/duckdb/pull/15637. Accessed: 2026-01-23.

[11] ClickHouse. 2022. ClickBench. https://github.com/ClickHouse/ClickBench. Original release: 2022-07-11. Accessed: 2026-01-23.

[12] Yann Collet. 2011. LZ4. https://github.com/lz4/lz4. Original release: 2011-03-25. Accessed: 2026-01-10.

[13] dtim-upc. 2021. NextiaJD. https://github.com/dtim-upc/NextiaJD. Original release: 2021-05-17. Accessed: 2026-01-23.

[14] Facebook. 2015. Zstandard. https://github.com/facebook/zstd. Original release: 2015-01-24. Accessed: 2026-01-10.

[15] Francesco Gargiulo. 2025. OnPair C++ Implementation. https://github.com/gargiulofrancesco/onpair_cpp. Accessed: 2026-07-01.

[16] Bogdan Ghita, Peter Boncz, and Duarte Tomé. 2019. Public BI Benchmark – Part 1. https://zenodo.org/records/6277287. Accessed: 2026-01-23.

[17] Google. 2011. snappy. https://github.com/google/snappy. Original release: 2011-03-07. Accessed: 2026-05-13.

[18] Paul Groß, Daniel ten Wolde, and Peter Boncz. 2019. Adaptive Factorization Using Linear-Chained Hash Tables. (2019).

[19] Dong He, Supun C. Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query Processing on Tensor Computation Runtimes. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2811–2825. https://doi.org/10.14778/3551793.3551833

[20] Sven Hepkema, Azim Afroozeh, Charlotte Felius, Peter Boncz, and Stefan Manegold. 2025. G-ALP: Rethinking Light-weight Encodings for GPUs. In *21st International Workshop on Data Management on New Hardware (DaMoN)*. ACM. https://doi.org/10.1145/3736227.3736242

[21] Allison L. Holloway, Vijayshankar Raman, Garret Swart, and David J. DeWitt. 2007. How to Barter Bits for Chronons: Compression and Bandwidth Trade Offs for Database Scans. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. 389–400.

[22] Matteo Interlandi, Nicolas Bruno, Brandon Haynes, Carlo Curino, Rathijit Sen, Yinan Li, et al. 2026. CoddSpeed: Hardware Accelerated Query Processing in Microsoft Fabric. In *Companion of the International Conference on Management of Data*. ACM, 359–372. https://doi.org/10.1145/3788853.3803077

[23] Violeta Kastreva, Philip Whittington, Dennis Komm, and Tiago Pimentel. 2026. Tokenisation over Bounded Alphabets is Hard. In *The Fourteenth International Conference on Learning Representations*. https://openreview.net/forum?id=Xhf9YqwlM4

[24] Maximilian Kuschewski, Dominik Sauerwein, Ammar Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26. https://doi.org/10.1145/3589263

[25] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage Using Both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data*. 311–326.

https://doi.org/10.1145/2882903.2882925
[26] Jigao Luo, Qi Chen, and Carsten Binnig. 2026. Do GPUs Really Need New Tabular File Formats?. In *22nd International Workshop on Data Management on New Hardware (DaMoN)*. ACM. https://doi.org/10.1145/3789237.3809125
[27] NVIDIA Corporation. 2024. *NVIDIA Blackwell Architecture Technical Brief.* https://resources.nvidia.com/l/en-us-blackwell-architecture
[28] Wesley Pace, Chen She, Lei Xu, et al. 2025. Lance: Efficient Random Access in Columnar Storage through Adaptive Structural Encodings. *CoRR* abs/2504.15247 (2025). https://doi.org/10.48550/arXiv.2504.15247 arXiv:2504.15247
[29] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, et al. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1080–1091.
[30] Alexander van Renen, Daniel Horn, Philipp Pfeil, et al. 2024. Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3694–3706. https://doi.org/10.14778/3681954.3682031
[31] Robin Vonk, Joost Hoozemans, and Zaid Al-Ars. 2025. GSST: Parallel String Decompression at 191 GB/s on GPU. In *Proceedings of the 5th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*. 8–14.
[32] Vortex Team. 2026. vortex-data/vortex: An Extensible, State-of-the-Art Columnar File Format. https://github.com/vortex-data/vortex. Accessed: 2026-02-12.
[33] Philip Whittington, Gregor Bachmann, and Tiago Pimentel. 2026. Tokenisation is NP-Complete. In *Tokenization Workshop*. https://openreview.net/forum?id=zGMXftuVZz
[34] Bowen Wu, Wei Cui, Carlo Curino, Matteo Interlandi, and Rathijit Sen. 2025. Terabyte-Scale Analytics in the Blink of an Eye. https://doi.org/10.48550/arXiv.2506.09226 arXiv:2506.09226 [cs.DB]
[35] Xuanhe Zeng, Rui Meng, Marcus Prammer, et al. 2025. F3: The Open-Source Data File Format for the Future. *Proceedings of the ACM on Management of Data* 3, 4 (2025), 1–27. https://doi.org/10.1145/3749163
[36] Jacob Ziv and Abraham Lempel. 1977. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343. https://doi.org/10.1109/TIT.1977.1055714
[37] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering*.

A STS_ℓ NP-HARDNESS

Theorem 1. *STS_ℓ is NP-hard for every fixed $\ell \geq 2$.*

Proof. We reduce from the NP-hard decision version of Maximum Vertex Coverage on bipartite graphs (VCB) [6]. Given a bipartite graph $G = (L \cup R, E)$ and positive integers K and q, decide whether there is a set $U \subseteq L \cup R$ with $|U| \leq K$ that covers at least q edges, i.e., at least q edges have at least one endpoint in U.

Let $m := |E|$ and let $\ell \geq 2$ be fixed. We construct an instance of STS_ℓ.

Introduce one character x_u for every $u \in L$, one character y_v for every $v \in R$, and one extra character z. For every edge $(u, v) \in E$, we create an edge gadget $x_u z^{(\ell-1)} y_v$ of length $\ell + 1$. Next we define

$$C := \{z^{(\ell-1)}\} \cup \{x_u : u \in L\} \cup \{y_v : v \in R\}.$$

The multiset $\mathcal{T}$ consists of all edge gadgets, and additionally $m + 1$ copies of each string $c \in C$ that we refer to as *copy strings*.

Finally, we set $K' := |C| + K$ and $B' := (m + 1)|C| + 3m - q$. We show that (G, K, q) is a yes-instance of VCB if and only if $(\mathcal{T}, K', B')$ is a yes-instance of STS_ℓ.

Forward direction. Suppose there is a set $U \subseteq L \cup R$ with $|U| \leq K$ covering $r \geq q$ edges. Define

$$D := C \cup \{x_u z^{(\ell-1)} : u \in L \cap U\} \cup \{z^{(\ell-1)} y_v : v \in R \cap U\}$$

with $|D| = |C| + |U| \leq |C| + K = K'$. Each of the $(m + 1)|C|$ copy strings is itself contained in D and can be encoded as a dictionary symbol at cost 1. Hence the copy strings contribute a total cost of $(m + 1)|C|$.

Next, consider the edge gadget $x_u z^{(\ell-1)} y_v$. If $u \in U$, then it can be encoded as $x_u z^{(\ell-1)}$ and y_v with cost 2. Similarly, if $v \in U$, it can also be encoded with cost 2. If neither u nor v are in U, the edge gadget can only be encoded as x_u, $z^{(\ell-1)}$, and y_v with cost 3. Therefore the r covered edges contribute cost $2r$, and the remaining $m - r$ edges contribute cost $3(m - r)$.

Thus,

$$\begin{aligned}\mathrm{cost}_D(\mathcal{T}) &= (m + 1)|C| + 2r + 3(m - r) \\ &= (m + 1)|C| + 3m - r \\ &\leq (m + 1)|C| + 3m - q = B',\end{aligned}$$

so the constructed STS_ℓ instance is a yes-instance. See Figure 7 for an illustration.

Reverse direction. Suppose there is a dictionary D with $|D| \leq K'$ and $\mathrm{cost}_D(\mathcal{T}) \leq B'$.

First, we show that $C \subseteq D$. For contradiction, assume that some $c \in C$ is not in D. Then, each of the $(m + 1)$ copies of c in $\mathcal{T}$ must be encoded with a cost of at least 2 instead of cost 1. Thus, the copy strings contribute cost at least $(m + 1)|C| + (m + 1)$.

Moreover, every edge gadget has length $\ell + 1$, while dictionary symbols have length at most ℓ. Therefore every edge gadget must be encoded with cost at least 2. Hence the total cost is at least

$$\begin{aligned}\mathrm{cost}_D(\mathcal{T}) &\geq (m + 1)|C| + (m + 1) + 2m \\ &= (m + 1)|C| + 3m + 1 \\ &> (m + 1)|C| + 3m - q = B',\end{aligned}$$

contradicting the assumption that $\mathrm{cost}_D(\mathcal{T}) \leq B'$. Thus $C \subseteq D$.

Since $C \subseteq D$ and $|D| \leq |C| + K$, there are at most K dictionary symbols not contained in C. As noted above, no edge gadget $x_u z^{(\ell-1)} y_v$ can be encoded with cost less than 2. Moreover, the copy strings contribute cost at least $(m + 1)|C|$.

Hence

$$\mathrm{cost}_D(\mathcal{T}) \leq B' = (m + 1)|C| + 3m - q$$

implies that the total cost of edge gadgets is at most $3m - q$, and therefore at least q edge gadgets must have cost exactly 2.

Every edge gadget $x_u z^{(\ell-1)} y_v$ with cost 2 is split into two dictionary symbols $x_u z^{(t)}$ and $z^{(\ell-1-t)} y_v$ for some $t \in \{0, \ldots, \ell - 1\}$. At least one of these symbols is not in C, since $\ell \geq 2$. Define

$$\begin{aligned}U := \{&u \in L : x_u z^{(t)} \in D \text{ for some } t \geq 1\} \\ &\cup \{v \in R : z^{(t)} y_v \in D \text{ for some } t \geq 1\}.\end{aligned}$$

Each element in $D \setminus C$ contributes to at most one vertex of U, therefore $|U| \leq K$. Moreover, every edge gadget with cost 2 has at least one endpoint in U. Since at least q edge gadgets have cost 2, the set U covers at least q edges.

Hence the original VCB instance is a yes-instance. This concludes the proof. □

B STS_ℓ WITH FIXED ALPHABET SIZE

Theorem 2. *STS_ℓ can be solved in polynomial time for every fixed ℓ and fixed alphabet size $n := |\Sigma|$.*

Proof. Let $(\mathcal{T}, K, B)$ be an instance over an alphabet Σ, where both ℓ and $n := |\Sigma|$ are fixed. Let $S := \bigcup_{i=1}^{\ell} \Sigma^i$ be the set of all strings over Σ of length at most ℓ. Since $|S| = \sum_{i=1}^{\ell} n^i \leq \ell n^\ell$ is constant, we can enumerate all $2^{|S|}$ subsets $D \subseteq S$ in constant time. For each such subset D, we check whether $|D| \leq K$; if so, we compute $\mathrm{cost}_D(\mathcal{T})$

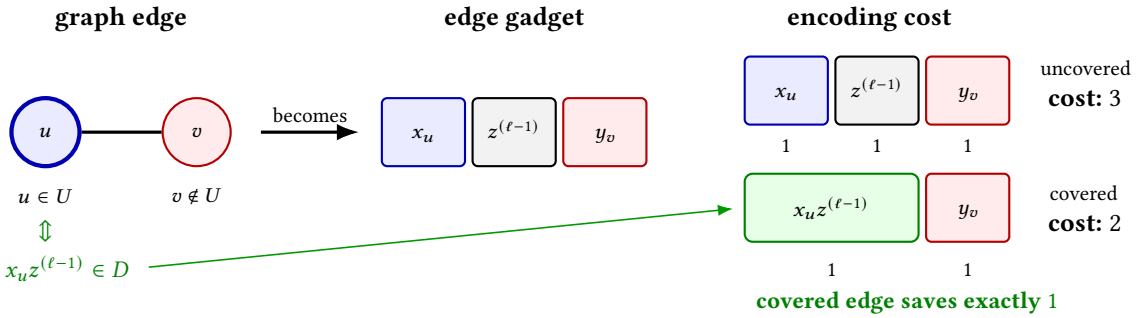


Figure 7: In the reduction, each graph edge (u, v) becomes the string gadget $x_u z^{(\ell-1)} y_v$. If $u \in U$, we add the length-ℓ string $x_u z^{(\ell-1)}$ to the dictionary D (and similarly $v \in U \Leftrightarrow z^{(\ell-1)} y_v \in D$), which lowers the encoding cost of the edge gadget from 3 to 2.

by dynamic programming in polynomial time. The instance is a yes-instance if and only if some enumerated subset D satisfies $|D| \leq K$ and

$$\text{cost}_D(\mathcal{T}) \leq B.$$

Thus the problem can be solved in polynomial time. □