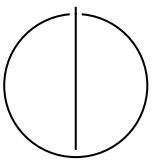
SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
DATA SCIENCE AND ENGINEERING

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Improved Symbol Table Construction for FSST Compression

Hedi Chehaidar



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — DATA SCIENCE AND ENGINEERING

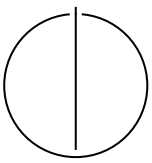
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Improved Symbol Table Construction for FSST Compression

Verbesserte Symboltabellenkonstruktion für die FSST-Komprimierung

Author:	Hedi Chehaidar, hedi.chehaidar@tum.de, 03783562
Examiner:	Prof. Dr. Thomas Neumann
Supervisors:	Prof. Dr. Andreas Kipf, Mihail Stoian
Submission Date:	13.02.2026



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 13.02.2026 Hedi Chehaidar

Acknowledgments

First of all, I want to thank God for the guidance and strength that he gave me to complete this thesis.

I would like to express my sincere gratitude to all those, who supported me morally and technically throughout this work.

I would like to thank my supervisor Mihail Stoian for the continuous guidance, insightful feedback, and trust in my ideas. The discussions we had, from discussing ideas to detailed implementation improvements, helped shape the thesis and improve its quality. I am thankful for the encouragements and for pushing me to always explore more optimization possibilities. I am grateful for the practical insights, constructive criticism, and technical expertise, especially regarding algorithmic correctness and performance evaluation. The feedback I received helped turn initial ideas into well-structured and measurable contributions.

I am grateful to the broader research and open-source community whose work laid the foundation for this thesis. The availability of high-quality implementations, documentation, and discussions around compression methods made it possible to build upon existing work and focus on meaningful improvements.
A special thanks goes to everyone who shared tools, datasets, or prior experience that helped accelerate experimentation and evaluation.

Finally, I want to thank my friends and family for their constant support, patience, and encouragement throughout this journey. Their motivation helped me stay focused and enthusiastic, especially during long debugging sessions and performance tuning phases.

This thesis would not have been possible without the collective contributions of all these people, and I am truly thankful for their support and inspiration.

Abstract

Modern analytical database systems process large volumes of string data, where efficient compression is essential for reducing memory footprint and improving performance. While the Fast Static Symbol Table (FSST) compression scheme provides excellent decompression speed and random-access capabilities, its effectiveness heavily depends on the quality of the constructed symbol table.

The original FSST algorithm relies on heuristic, greedy symbol selection and corpus compression, which can lead to suboptimal symbol choices and limit achievable compression factors.

This thesis presents several enhancements to the FSST compression method that improve symbol table construction and full corpus compression while preserving FSST's core advantages. The central contribution is a refined symbol selection process that systematically identifies more effective symbols and avoids redundant or conflicting choices.

First, a dynamic programming approach is introduced to evaluate and select higher-quality symbols within each generation, in addition to improving the corpus compression after the table construction. Second, an additional frequency counter is incorporated to accelerate the discovery of longer symbols and to explicitly favor them in subsequent generations, improving the exploitation of longer recurring patterns in the data. Third, a symbol pruning mechanism is applied to eliminate conflicting and redundant symbols, ensuring a more compact and effective symbol table.

Together, these techniques significantly improve the robustness and quality of the symbol table generation process, as well as enhance the used compression method. Experimental evaluation demonstrates that the proposed contributions lead to consistently improved compression factors with an average of 9.6% compared to the original FSST algorithm, while maintaining its fast decompression and random-access properties. The results show that careful algorithmic refinement of symbol selection can yield substantial gains without altering the lightweight and practical nature of FSST, making the improved approach well suited for use in modern analytical systems.

Contents

1 Introduction

1.1 Motivation

Modern data management systems increasingly operate on large-scale, string-heavy datasets. In analytical databases, strings are omnipresent and appear in many forms, including URLs, file paths, identifiers, log messages, categorical attributes, and semi-structured data originating from web, cloud, and enterprise applications. Studies of real-world database workloads show that string columns often constitute a substantial fraction of the overall data volume in terms of storage [1]. Prior analyses of analytical benchmarks and production systems report that strings can account for a significant portion of columnar storage, frequently dominating memory usage in dictionary-encoded or compressed representations [2], [3].

The growing prevalence of string data places strong demands on compression techniques used in analytical systems. Effective compression reduces memory footprint, improves cache utilization, and lowers memory bandwidth pressure, all of which are critical for high-performance query processing. At the same time, analytical workloads require fast random access to individual values, as queries often scan and decode only a subset of columns or rows. This combination of requirements makes general-purpose, block-based compressors such as Zstandard [4] or LZ4 [5] less suitable despite their strong compression factors, as they typically require decompressing entire blocks before accessing individual strings.

To address this gap, lightweight string compression schemes have been proposed that prioritize fast decompression and random access. One prominent example is the FSST compression algorithm [6]. FSST compresses strings by replacing frequent byte sequences with compact symbols from a statically constructed symbol table. During decompression, symbols can be expanded independently, allowing direct access to individual strings without scanning neighboring data. As a result, FSST achieves decompression speeds that are competitive with, and often superior to, more heavyweight compression schemes, making it attractive for use in modern analytical databases.

However, the compression effectiveness of FSST critically depends on the quality of

its symbol table. The original FSST algorithm constructs this table using a greedy approach that iteratively selects symbols based on local heuristics. While this strategy is computationally efficient and aligns with FSST's design goal of lightweight processing, it can lead to suboptimal symbol choices. In particular, greedy selection may favor short or locally frequent symbols that conflict with longer or more informative sequences, therefore limiting the discovery of beneficial longer symbols, or introducing redundancy within the symbol table. Furthermore, once a symbol is selected, its impact on future generations of symbols is not globally optimized, which can prevent the algorithm from converging toward a symbol set that maximizes overall compression gain.

These limitations suggest that there is room for improvement in FSST's symbol table construction without compromising its core advantages. By revisiting the greedy nature of symbol selection and incorporating more informed decision-making into the construction process, it is possible to improve compression factors by up to 47.7% while retaining FSST's fast decompression and random-access properties. This thesis explores such improvements, focusing on enhanced symbol selection strategies that address the shortcomings of the original greedy approach.

1.2 Thesis Outline

The rest of the thesis is structured as follows:

Chapter 2 (Background) introduces the fundamental concepts required to understand the techniques developed in this thesis. It begins with an overview of dynamic programming and contrasts it with greedy algorithmic approaches, highlighting their respective strengths and limitations. The chapter then introduces the core data structures used throughout the thesis, namely tries and max heaps, which play a central role in symbol generation and selection.

Chapter 3 (Related Work) reviews existing work in the area of data compression. It first discusses widely used block-based compression algorithms such as LZ4 and Zstandard, explaining their general design principles and trade-offs. The chapter then presents the original FSST compression algorithm, detailing its symbol table construction and encoding process, which form the basis for the improvements proposed in this thesis.

Chapter 4 (Approach) describes the main contributions of this work. It introduces three enhancements to the FSST symbol table construction process: a dynamic programming-based method for improved symbol selection and full corpus encoding, the introduction of a third frequency counter to accelerate the discovery and prior-

itization of longer symbols, and a symbol pruning strategy to eliminate conflicting and redundant symbols. Each technique is explained in detail and integrated into the overall compression pipeline.

Chapter 5 (Evaluation) evaluates the proposed enhancements experimentally. It presents the datasets, experimental setup, and benchmarking methodology used in the evaluation. The chapter analyzes the impact of the proposed techniques on compression factors and runtime, comparing the improved FSST variants against the original algorithm and when incorporating the enhancements into DICT FSST [7], a compression method that uses FSST internally.

Chapter 6 (Discussion & Future Work) reflects on the achieved improvements and discusses directions for future work, with emphasis on potential optimization opportunities to reduce the runtime overhead introduced by the enhanced symbol selection techniques.

Chapter 7 (Conclusion) summarizes the contributions and findings of the thesis.

2 Background

2.1 Dynamic Programming vs. Greedy

Many algorithmic problems involve selecting a sequence of decisions that together optimize a global objective. Two widely used approaches for solving such problems are greedy algorithms and dynamic programming (DP). While both aim to construct efficient solutions, they differ fundamentally in how they explore the solution space and reason about optimality.

2.1.1 Greedy Algorithms

Greedy algorithms build a solution incrementally by making locally optimal decisions at each step. At every stage, the algorithm selects the option that appears best according to a predefined heuristic, without reconsidering previous choices. This strategy is attractive due to its simplicity, low computational overhead, and ease of implementation.

Greedy approaches are particularly effective when a problem exhibits the greedy-choice property, meaning that a locally optimal decision can be shown to lead to a globally optimal solution.

Example. An easy example illustrating the greedy-choice property is the problem where we are given a sequence of distinct numbers and we want to take a subset of K numbers with the goal of maximizing the sum of those numbers. The intuitive and correct way to choose the numbers is by selecting the largest K numbers of the sequence for the subset (denoted by S). We can prove this by contradiction: suppose there is an optimal solution where a number of the chosen subset does not belong to S, we can swap that number with an unchosen number of the set S and we would have a strictly better solution, which leads to contradiction.

Despite this limitation, greedy algorithms are frequently used in performance-critical systems where execution speed and simplicity are prioritized over absolute optimality.

2.1.2 Dynamic Programming

Dynamic programming (DP) addresses optimization problems by systematically exploring all relevant subproblems and combining their solutions to obtain a globally optimal result. The core idea is to decompose a problem into overlapping subproblems, solve each subproblem once, and store its result to avoid redundant computation.

Unlike greedy algorithms, dynamic programming evaluates the long-term consequences of decisions. By considering multiple possible choices at each stage and selecting the one that minimizes or maximizes a well-defined cost function, DP can guarantee optimality when the problem satisfies optimal substructure and overlapping subproblems.

The primary trade-off of dynamic programming is computational complexity. DP solutions often require more time and memory than greedy alternatives, especially when the state space is large. As a result, practical DP implementations frequently rely on carefully designed cost models, state compression, or bounded problem sizes to remain efficient.

2.2 Data Structures

Efficient algorithmic design relies not only on the choice of optimization strategy but also on the use of appropriate data structures. This section introduces the two core data structures used throughout this thesis: max heaps and tries. Both play an important role in managing candidates and efficiently representing string data.

2.2.1 Max Heap

A max heap is a complete binary tree-based data structure that maintains the heap property: the key of each node is greater than or equal to the keys of its children. As a result, the maximum element is always stored at the root of the heap.

The primary operations supported by a max heap include insertion, extraction of the maximum element, and key updates. Each of these operations can be performed in logarithmic time with respect to the number of elements in the heap. This efficiency makes max heaps particularly well suited for priority-based selection tasks, where the most valuable or promising element must be accessed repeatedly.

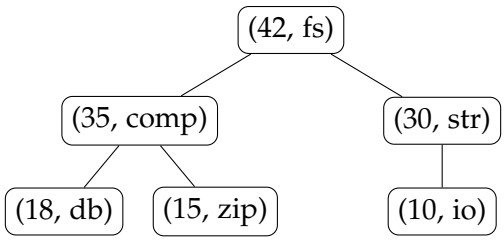


Figure 2.1: Initial max heap.

Example. The max heap from the example above stores the following data: {(42, “fs”), (35, “comp”), (30, “str”), (18, “db”), (15, “zip”), (10, “io”)}. These (int, string) pairs can represent gains and symbols respectively in the context of FSST, where the gain is a heuristic to measure how good a symbol is. Storing the information in this format allows for the symbol with the highest gain to be selected first from the max heap with the ’pop’ operation, extracting the element in the root node.

As the figure shows, each node has a higher key than its children. In this case the key comparison occurs between the integers (the first element of the pair) then between the strings in case of a tie.

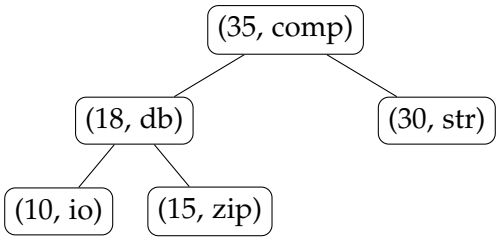


Figure 2.2: Max heap after pop operation.

The pop operation extracts the element of the root node, which is the element with the highest key. Then the rest of the elements are reordered to maintain the max heap property. This operation has a runtime complexity of $O(\log(n))$, where n is the number of nodes in the heap.

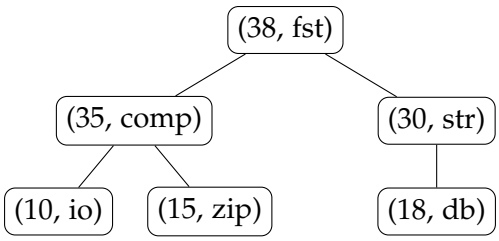


Figure 2.3: Max heap after inserting the element (38, fst).

The insert operation inserts a new element to the heap and then reorders the rest of the elements to maintain the max heap property like in the pop operation. This operation also has a runtime complexity of $O(\log(n))$.

In the context of algorithmic optimization, max heaps are often used to manage candidate sets ordered by a score or heuristic value. They enable fast retrieval of the currently best candidate while allowing dynamic updates as new candidates are generated or existing ones are reweighted.

2.2.2 Trie

A trie, also known as a prefix tree, is a tree-based data structure used to store and retrieve strings efficiently by exploiting their shared prefixes. Each node in a trie represents a prefix of one or more strings, and edges correspond to individual characters or bytes. Strings are represented by paths from the root to terminal nodes which can also store other data related to the strings (like mappings).

Tries provide efficient operations for prefix-based queries, insertion, and lookup, all of which can be performed in time proportional to the length of the string rather than the number of stored strings. This makes them particularly suitable for applications involving multiple search operations for strings with common prefixes.

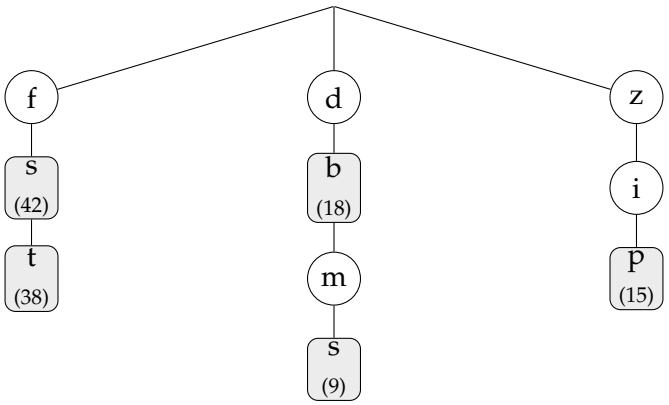


Figure 2.4: Example trie storing short strings.

Example. The trie from the example above stores the 5 strings “fs”, “fst”, “db”, “dbms”, and “zip” with a mapping for each string as an integer stored in the corresponding terminal node. The strings stored can be interpreted as the symbols of a symbol table and the integers as the corresponding codes (one code for each symbol) in the context of FSST. Details will follow in next sections.

As the trie shows, terminal nodes (gray nodes) mark the end of a symbol and contain the corresponding code. A symbol can be searched by a traversal of the trie from the root to the terminal node. Each edge corresponds to the next character in the symbol search.

Symbols can have other symbols as a prefix and in that case the terminal node of the prefix symbol will be an ancestor of the terminal node of the bigger symbol. For example, the trie above stores the symbol “db” which is a prefix of the symbol “dbms”.

3 Related Work

3.1 Block-Based Compressors

General-purpose block-based compression algorithms are widely used across storage systems, operating systems, and data processing pipelines due to their ability to achieve high compression factors on diverse data types. These compressors operate by partitioning the input data into blocks of fixed or variable size and compressing each block independently. This design enables parallel compression and decompression, robust error containment, and adaptability to different data distributions.

3.1.1 LZ4

One of the most prominent examples of such algorithms is LZ4 [5], a lightweight compressor based on the Lempel-Ziv 77 (LZ77) family of dictionary compression techniques. The general LZ77 technique works by scanning an input block (usually 128 KB), detecting byte sequences that appeared before. Repeated sequences are encoded as a pair (`offset`, `length`) where `offset` references the first appearance of the sequence in the output stream.

What distinguishes LZ4 is that during compression a hash table is used to quickly identify matches in the input and greedily accepting the first match found. This approach minimizes CPU usage, providing compression speed > 500 MB/s per core.

LZ4 decompression is close to just performing memory copy ('memcpy') operations, making it bounded by memory bandwidth with speed of multiple GB/s per core, typically reaching RAM speed limits on multi-core systems.

LZ4 prioritizes decompression speed over compression factors, making it suitable for performance-critical workloads where fast data access is required. It achieves compression by identifying repeated byte sequences within a block and encoding them as references to earlier occurrences.

While LZ4 offers fast decompression, its relatively small window size and simple matching strategy limit its ability to exploit longer-range redundancies, especially across block boundaries.

3.1.2 Zstd

Zstandard (Zstd) [4] represents a more advanced block-based compression approach developed by Facebook that balances compression factor and performance. Zstandard combines LZ-style dictionary matching with entropy coding and supports large compression windows, allowing it to capture long-distance repetitions within a block. Zstd encodes unmatched byte sequences, called literal bytes, in addition to the offset and length attributes of match pairs using Finite State Entropy (FSE) and Huffman coding techniques. As a result, Zstandard often achieves significantly higher compression factors than lightweight compressors such as LZ4, particularly on structured or repetitive data. However, this improved compression comes at the cost of increased decompression complexity and reduced random-access efficiency, as individual values typically cannot be decompressed without processing the surrounding block.

3.1.3 Limitations

While block-based compressors are highly effective for bulk storage and sequential access patterns, their design fundamentally limits random access performance. In analytical database systems, queries often access individual values or small subsets of columns rather than scanning entire blocks. In such scenarios, block-based compression requires decompressing full blocks even when only a small portion of the data is needed, leading to unnecessary computation and memory traffic. This limitation has been highlighted in prior work on column-oriented storage and analytical query processing, where fine-grained access and low-latency decompression are critical for performance.

Several systems attempt to mitigate this issue by using smaller block sizes or by combining block-based compression with dictionary encoding. However, reducing block size typically degrades compression efficiency, while hybrid approaches increase system complexity. As discussed in the FSST paper, these trade-offs make block-based compressors less attractive for scenarios where fast random access to compressed strings is a primary requirement [6].

Consequently, lightweight compression schemes that avoid block-level dependencies have gained attention in the context of analytical databases. These schemes trade some compression efficiency for predictable decompression costs and fine-grained access. FSST belongs to this class of compressors, providing symbol-based compression that allows individual strings to be decompressed independently. Understanding the strengths and limitations of block-based compressors is therefore essential for positioning FSST and the improvements proposed in this thesis relative to existing compression techniques.

3.2 FSST

3.2.1 General Idea

Fast Static Symbol Table (FSST) [6] is a lightweight string compression algorithm that works by replacing substrings (symbols) in the original data with one-byte codes. A symbol table is constructed from a sample of the data, mapping symbols to codes.

The term "static" refers to the fact that any string can be compressed or decompressed independently using the symbol table, without requiring any prior context, unlike block-based compressors discussed in the previous section.

Individual strings in databases are typically not larger than 200 bytes, with the majority being less than 30 bytes [6]. This makes the random-access property of FSST more valuable compared to block-based compressors, which often need to decompress an entire block of 128 KB to retrieve a single string.

FSST also supports compressed query processing, since equality between original strings is equivalent to equality between their compressed representations. It may also be possible to perform more complex operations on compressed strings, such as pattern matching, which can reduce the need to decompress strings when processing a wide range of queries.

Symbols have lengths ranging from 1 to 8 bytes, and each symbol is mapped to a code from 0 to 254, with code 255 being reserved as an escape code that plays an important role during decompression. Consequently, the total number of usable symbols is 255. This results in a worst-case symbol table memory overhead of $8 \times 255 + 255$ bytes (symbols and their lengths), which is very unlikely in practice, as the average symbol length is usually close to 2 bytes [6].

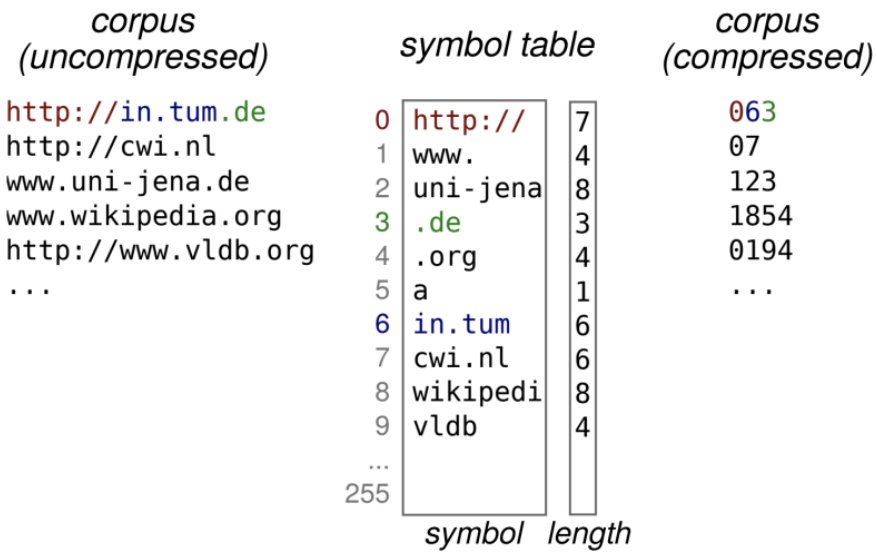


Figure 3.1: FSST compression example [6]

Example FSST Compression. Figure 3.1 shows the first 10 symbols of an example symbol table, with an additional array for symbol lengths. For example, the first string is compressed into three bytes (codes), where each code is the index of the corresponding symbol in the symbol table.

3.2.2 Decompression

FSST decompression is fairly simple. The algorithm iterates over the encoded input stream byte by byte and appends the corresponding symbol from the symbol table to the output stream. The exception of the escape code (code 255) can be checked with an `if` statement, where in that case the algorithm appends the following raw byte after the escape code as is and continues with the next byte.

The original FSST open-source implementation [8] optimizes decoding to avoid performing an `if` statement for every byte and instead loads 4 bytes every time and then checks if they contain the escape code. FSST decoding is therefore comparably fast approaching 2 GB/s according to the original paper's evaluation.

The improvements of this thesis do not affect the FSST decompression method in any way. Decompression was used in this thesis solely to validate correctness of the added contributions.

3.2.3 Compression

FSST compression is performed independently for each string in the corpus. A loop iterates over the bytes of the string. In each iteration, the longest matching symbol starting at the current position is used for encoding, and the current position in the string is updated accordingly. If no symbol in the table matches the current position, the byte is escaped by appending the escape code to the output stream, followed by the raw byte from the string.

The function used to find the longest matching symbol is called `findLongestSymbol`, and its implementation is shown in Lst. 3.1. The same function is also used during symbol table construction.

This greedy approach enables fast encoding at the cost of a suboptimal compression factor. An alternative approach is presented in Section 4.1.

3.2.4 Symbol Table Construction

The construction of the symbol table is the most important component of the FSST algorithm with respect to the achieved compression factor, as the encoding quality is tightly dependent on the set of symbols selected for the table.

Selecting symbols is challenging because their effectiveness depends on one another whenever they overlap. This is the dependency issue described in the original paper. Therefore, the only reliable way to assess the significance of a symbol is to compress the sample using the current symbol table. However, evaluating all possible symbol tables is not feasible due to the enormous number of possibilities ($\binom{8N}{255}$ where N denotes the sample size in bytes).

Listing 3.1: FSST symbol table construction [6].

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

  def findLongestSymbol(st, text):
    var letter = ord(text[0])
    for code in range(st.sIndex[letter],
      st.sIndex[letter+1])
      if(text.startswith(st.symbols[code]))
        return code
    return letter

  def compressCount(st, count1,
    count2, text):
    var pos = 0
    var prev, code
    code = st.findLongestSymbol(text[pos:])

    while ((pos += st.symbols[code].len())
      < text.len())
      prev = code
      code = st.findLongestSymbol(
        text[pos:])
      count1[code]++
      count2[prev][code]++
      if (code >= 256)
        nextByte = ord(text[pos])
        count1[nextByte]++
        count2[prev][nextByte]++

  def buildSymbolTable(st, text):
    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

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

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

def makeIndex(st):
  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
  return st
```

The Original Idea's Algorithm. The FSST algorithm shown in Lst. 3.1 constructs the symbol table by iterating over a sample of the corpus for a fixed number of generations. In each generation, the current table is refined by discovering new symbol combinations and retaining those with the highest static gain. The gain of a symbol is computed as $gain = length \times frequency$, where length is the symbol length (between 1 and 8 bytes) and frequency is the number of times the symbol is selected when compressing the sample using the previous table.

In the original implementation, the number of generations is set to five and the sample size to 16 KB. The sample is processed incrementally, such that in each generation a larger fraction of the sample is considered for symbol selection.

The `buildSymbolTable` function is the main component of the algorithm, in which the five iterations over the sample are performed. Two counters, `count1` and `count2`, are used to record symbol frequencies when compressing the text using the previous symbol table. The counter `count1` tracks individual symbols, while `count2` records the concatenation of two symbols or the concatenation of a symbol and a literal byte.

The indices of these counters range from 0 to 511. Indices from 0 to 255 correspond to escaped bytes (literal bytes), while indices from 256 to $256 + st.nSymbols - 1$ (up to 511) denote actual symbols.

In each generation, the counters are populated using the `compressCount` function, which compresses the text with the current symbol table. This function counts individual symbols in `count1` and records all pairs of successive symbols, as well as combinations of a symbol followed by a literal byte, in `count2`.

Real symbols are stored in `st.symbols` starting from index 256 in alphabetical order. Symbols that begin with the same character are sorted in decreasing order of length. As a result, `st.findLongestSymbol` terminates after finding the first matching symbol.

A new symbol table is then constructed from the counters using the `makeTable` function. Each symbol, literal byte, and their concatenations are inserted into a max heap, where the key is the computed gain of the candidate symbol. The 255 symbols with the highest static gains are selected for the next symbol table. Finally, the `makeIndex` function sorts the symbols as described above and initializes the `st.sIndex` table, which stores the position (code) of the first symbol beginning with a given letter.

FSST Evolution. The FSST algorithm evolved from the previously described version to a branchless implementation that enables the use of SIMD instructions (AVX-512), significantly reducing encoding time. The primary bottleneck in encoding is the `findLongestSymbol` function. Therefore, a redesign of the symbol storage was introduced as a first optimization.

Symbol codes are now stored in two separate data structures: a 256×256 array

named `shortCodes` for symbols of length one and two bytes, and a hash table `hashTab` of size 1024 (1 KB) for symbols with lengths ranging from three to eight bytes. A lossy perfect hashing scheme is employed for longer symbols, using a hash function based on the first three bytes of the symbol. If inserting a symbol would result in a collision, the symbol is discarded, retaining only the symbol with the highest gain among colliding symbols of length three or greater.

For each character A for which no entry in `shortCodes[A][*]` is defined, the corresponding single-byte symbol is inserted if it exists. This guarantees that an access to `shortCodes` always returns the longest matching symbol of length one or two. The `findLongestSymbol` function first queries the `hashTab` to search for a matching symbol, then consults `shortCodes` if no match is found, and finally falls back to returning the literal byte. The function's return value is computed using a conditional move (MOV) instruction, thereby avoiding branching.

Limitation. Although the original FSST algorithm is highly optimized for runtime, particularly when using the SIMD variant, the greedy strategy employed by the `findLongestSymbol` function does not always select the optimal symbol, both during compression and during symbol table construction. This leaves room for improving the compression factor by adopting a different optimization strategy that more carefully considers conflicts between overlapping symbols, as discussed in the following section.

4 Approach

4.1 Dynamic Programming

4.1.1 Idea and Formula

The first contribution of the thesis to the FSST algorithm of Lst. 3.1 is replacing the `st.findLongestSymbol` method with a DP function that gives "one" best symbol given the position in the text to encode (or training text in `compressCount`) with a single table lookup in $O(1)$ runtime. This table called `opt` is constructed simultaneously with the DP table.

Initial Formula. To understand the utility of the used DP function in the code, let us first define the more intuitive DP function for $0 \leq i \leq n$ (where n is the text length) as:

$$dp[i] = \text{smallest compressed size of the first } i \text{ bytes of the given text.}$$

The result for the whole text is therefore stored in $dp[n]$. The base case is $dp[0] = 0$ as the compression of an empty string is always an empty string. The recurrence is defined for each other value as:

$$dp[i] = \min(2 + dp[i-1], 1 + \min_{\substack{0 \leq j < i \\ s[j:i] \in symbols}} dp[j]) \tag{4.1}$$

where `s[j:i]` denotes the substring of the text from position j to $i-1$ (0-indexed). The formula simply chooses between either escaping the current byte or selecting one symbol that minimizes the DP value. In the first case, an overhead of two bytes must be added caused by the escape byte followed by the raw byte from the string during encoding. In the second case, one additional byte is emitted for the code of the chosen symbol. All candidate symbols are the ones that match a suffix of the first i bytes, thus the constraints under the second *min*.

However, this formulation is not suitable for constructing `opt` on the fly: it iterates over symbols that end at position i, while `opt[i]` must store the best symbol that starts at position i.

Final Formula. Therefore, the following DP function is used instead, defined for $0 \leq i \leq n$ as:

$$dp[i] = \text{smallest compressed size of the last } n - i \text{ bytes of the given text.}$$

This is equivalent to saying that $dp[i]$ is the smallest compressed size of the suffix of the given text starting at position i (0-indexed). The result for the whole text is therefore stored in $dp[0]$. The base case is $dp[n] = 0$ as the suffix from position n is an empty string. The recurrence is defined for each other position as:

$$dp[i] = \min(2 + dp[i+1], 1 + \min_{\substack{i<j\leq n \\ s[i:j]\in symbols}} dp[j]). \tag{4.2}$$

The two options to choose from are the same as in the previous formula, but candidate symbols for the second option must now begin at position i as we are now calculating the results for suffixes instead of prefixes of the given text.

Equivalence. This DP problem is a segmentation problem, where each segment is either a symbol, the escape code or a raw byte of the string that must follow the escape code. All segments have the same cost and we want to minimize the number of segments used to compress a given text. The first formula builds the solution forwards (from index 0 to n), while the second formula builds the solution backwards (from index n to 0). Both formulations yield the same optimal cost for the whole text, as they solve the same problem, although the sequence of chosen symbols may differ in both approaches.

4.1.2 Example

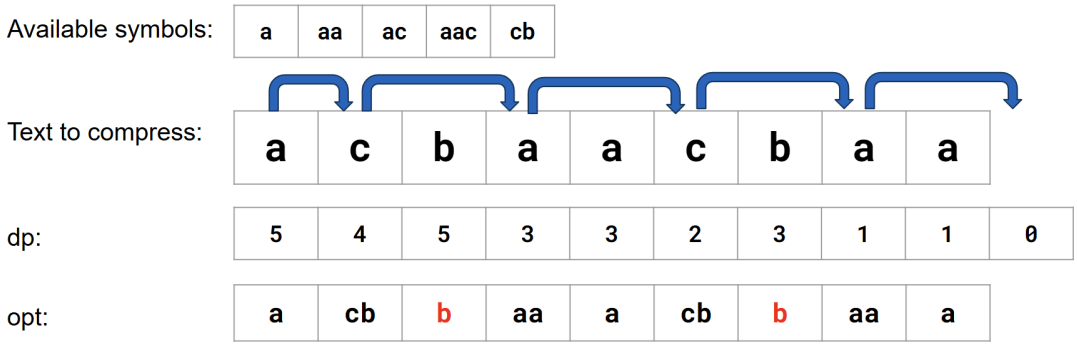


Figure 4.1: Example DP compression.

Figure 4.1 shows an example for compressing a text using the DP formula of Eq. (4.2). `dp[0]` indicates that we needed 5 bytes to compress the full text given the available symbols. `opt` contains one best symbol that yields to the best compression of the suffix beginning from the corresponding position. The values for both tables are filled backwards (from right to left) simultaneously according to the used formula. The arrows show the steps to recover the symbols used for compression, needed for counting frequencies in the `compressCount` function.

The greedy approach of the `findLongestSymbol` function would need 8 bytes to compress the same text with the same set of symbols. A difference can be seen in the first chosen symbol: `findLongestSymbol` would choose "ac" as the first symbol, which will lead to skipping the next byte as there are no symbols beginning with "b". On the other hand, the DP function chooses the symbol "a" which is not the longest possible symbol to choose, but it allows for the smallest final compression size.

4.1.3 Implementation

```
n = len(data)
# data is the given text to encode, or to use for compressCount
self.dp = [0] * (n+1)
self.opt = [0] * n
for i in reversed(range(n)):
  self.opt[i] = data[i]
  self.dp[i] = self.dp[i+1] + 2
  # start is the index of first symbol beginning with the byte at i
  start = self.sIndex[data[i]]
  end = self.sIndex[data[i] + 1]
  for code in range(start, end):
    sym = self.symbols[code]
    L = len(sym)
    if (
      i + L <= len(data)
      and self.dp[i] > 1 + self.dp[i + L]
      and data[i:i + L] == sym
    ):
      self.dp[i] = 1 + self.dp[i + L]
      self.opt[i] = code
```

Listing 4.1: DP construction for `dp` and `opt` implementing Eq. (4.2).

The code in Lst. 4.1 represents the python implementation of the DP function `st.buildDP` that fills both the DP and the `opt` tables in a single pass.

As the code shows, the DP table is filled from n to 0 in descending order, where for each position other than n the DP value is initialized with the corresponding value for the option of escaping the current byte. Then the inner loop iterates over the candidate symbols that begin with the current byte and match the next chunk of the text `data`. This code uses the `st.sIndex` table, whose construction was shown in Lst. 3.1. As a result, the iteration over the symbols is in decreasing order of their lengths. As a tie-breaker between symbols that yield the same DP value, this implementation chooses a symbol with the highest length (i.e., it occurs first in the iteration), thus the use of "$>$" instead of "$\geq$" in the `if` statement for updating the DP and `opt` tables (line 15). There is an exception to this case, when choosing to escape the byte also yields the smallest DP value, then no symbol will be chosen.

Trie Structure. This approach requires performing a memory comparison of at most 8 bytes for every symbol that begins with a certain byte. To avoid that, a trie is used in the C++ implementation of the contributions [9] to store the symbols in addition to the `hashTab` and `shortCodes` arrays. The trie is implemented as a vector of `TriNodes`, whose structure is shown in Lst. 4.2.

```
1 struct TrieNode {
2   int symbolCode; // code of a symbol ending here, -1 if none
3   int child[256]; // child indices, -1 if absent
4   TrieNode() : symbolCode(-1) {
5     for (int i = 0; i < 256; i++) child[i] = -1;
6   }
7 };
```

Listing 4.2: Trie node used to store FSST symbols for DP-based lookup.

Each node stores the code of the symbol ending at that node and the indices for the children nodes, where the maximum number of children is 256 (an edge for every possible byte). The maximum number of nodes in the trie is $(8 * 255 + 1)$, as all symbols in the worst case don't share prefixes and there are at most 255 actual symbols, with an additional node for the root. This makes the additional trie memory overhead around $2MB$ in the extreme case.

Now finding candidate symbols can be done efficiently by traversing the trie and comparing one byte at each iteration instead of a whole symbol, with the possibility to break from the inner loop when there are no symbols beginning with a certain prefix. The inner loop of the final implementation is shown in Lst. 4.3.

```
1  // walk trie for real symbols (1..8 bytes)
2  int node = 0;
3  int limit = min(Symbol::maxLength, n - (size_t)i);
4  for (int off = 0; off < limit; ++off) {
5    u8 bb = data[i + off];
6    node = trie[node].child[bb];
7    if (node == -1) break;
8    int code = trie[node].symbolCode;
9    if (code != -1) {
10     u32 L = off + 1;
11     // real symbol always emits 1 byte
12     u32 cost = 1 + dpCost[i + L];
13     if (cost <= bestCost) {
14       bestCost = cost;
15       bestCode = code;
16     }
17   }
18 }
```

Listing 4.3: Trie traversal in `st.buildDP`.

This trie traversal from the root implies iterating over the symbols in increasing order of their lengths. Thus the use of "$\leq$" in the `if` statement for updating the DP value (line 12), in order to keep the same tie-break logic as in the python implementation. However in this case choosing a symbol will be preferred over escaping the byte, when both options result in the same DP value, which can be helpful in later generations to discover more symbol combinations.

A similar traversal is used for adding each symbol into the trie, where instead of breaking when `node == -1`, a new node is created and the child entry is updated accordingly. Here, breaking is faster in practice than continuing the 8 full iterations without branching, as in most cases no symbols will be found after the second or third iteration, due to the low average length of symbols in the symbol table.

Having the trie as the structure used for the `findLongestSymbol` function, there is no need for the `sIndex` table and therefore the `makeIndex` function. As a result, including this method in the original C++ implementation of FSST [8], that already does not use `makeIndex`, becomes easier.

After building the DP and `opt` tables, choosing the next symbol in `compressCount` or in the compression function can be done by a simple lookup in the `opt` table. The implementation wraps this operation in a function called `findBestSymbol`.

4.2 3^{rd} Counter

The DP strategy introduced in the previous section replaces the greedy approach of the `findLongestSymbol` function. As a result, shorter symbols may become more valuable and therefore more frequent in terms of their counts computed by the `compressCount` function. To encourage the symbol selection method to still choose longer symbols, a third frequency counter `count3` is introduced. This counter keeps count of the appearances of all triples of consecutive symbols used for compressing the sample, in addition to the triples constructed from a pair of symbols followed by the next literal byte in the text data.

Counting triples of symbols will make converging the symbol table faster, and with combination of the DP approach of counting frequencies, more variation is introduced to candidate symbols from one generation to another. That means, adding a third counter while still having `findLongestSymbol` instead of the DP function will simulate skipping generations and quickly converging the symbol table to have long symbols that are concatenated from symbols of the first couple generations. The DP approach, on the other hand, balances the symbol selection by looking at the overall compression result and considering symbols that globally achieve the best compression, regardless of their lengths. Therefore, this combination improves the overall symbol selection.

Integration into FSST. This contribution affects both the `compressCount` and the `makeTable` functions. `compressCount` should now have a third variable (for example `prev2`), in addition to `code` and `prev` that were used in the implementation in Lst. 3.1, to store the code chosen two steps ago. Then the triples, whose frequencies should be increased are (`prev2`, `prev`, `code`) and (`prev2`, `prev`, `nextByte`), in case `code` is an actual symbol and not a literal byte.

The function `makeTable` should also consider all possible three-symbol concatenations as potential candidates and push them to the heap, with the gain being calculated as usual.

Storing `count3` as a 3-dimensional table, analogous to `count1` and `count2`, would be highly inefficient: it would require 512^3 entries and iterating over all three-symbol concatenations will be extremely slow. Noticing that we only need to consider triples having non-zero counts (i.e., they actually appear in the sample compression) makes it possible to store the triples in a hash map. The size of `count3` will be then linear to the sample size, same as `count1` and `count2`.

```
struct Count3 {
  unordered_map<u32, u16> m;
  void clear() { m.clear(); }
  void inc(u16 a, u16 b, u16 c) {
    u32 k = pack3(a,b,c);
    auto it = m.find(k);
    if (it == m.end()) m.emplace(k, 1);
    else it->second++;
  }
  u16 get(u16 a, u16 b, u16 c) const {
    auto it = m.find(pack3(a, b, c));
    return it == m.end() ? 0 : it->second;
  }
};

static inline u32 pack3(u16 a, u16 b, u16 c) {
  // each code fits in 9 bits (0..511)
  return ((u32)a << 18) | ((u32)b << 9) | (u32)c;
}
```

Listing 4.4: Count3 structure.

The code snippet in Lst. 4.4 shows the `count3` implementation in the C++ codebase of Ref. [9]. As the codes take at most 9 bits each, the triple can be packed into 27 bits and therefore wrapped in an unsigned integer variable of 32 bits. This allows fast frequency addition and frequency query of a specific triple.

4.3 Symbol Pruning

4.3.1 Motivation

Recall the baseline symbol selection in Lst. 3.1. The `makeTable` function greedily takes the first 255 symbols in terms of static gain, without taking into account the conflicts between those symbols, and leaving most of the candidates in the heap unchosen. Especially that now the number of candidates grew by around $\times 1.5$ due to the third frequency counter, there is more potential to miss better symbols. Some of the chosen symbols may be a substring of other bigger symbols and therefore conflicting with them. It means that in `compressCount` of the next generation, the bigger symbol is most likely to be chosen over the smaller one. This possibility is higher when the smaller one is a prefix and it is certain when using `findLongestSymbol` instead of the DP function.

Then the new count of the smaller symbol will be smaller after the next generation than its current count. Therefore, one role of a generation, in addition to discovering new symbol combinations through concatenations, is correcting this overestimation in the counts of smaller symbols.

Symbol pruning is one step forward to this goal. By partially correcting the counts of symbols that are parts of other bigger symbols, we save time in the next generation and add more variety to the symbols chosen in the symbol table.

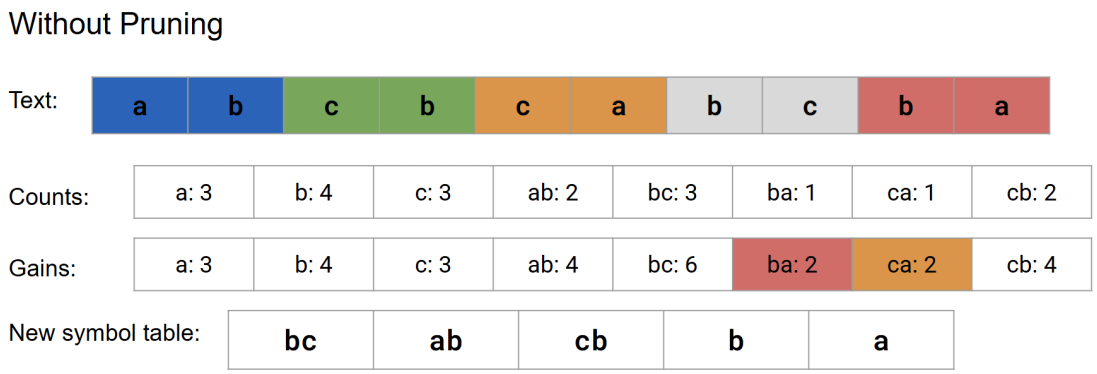


Figure 4.2: Example of symbol selection without pruning.

Example without Pruning. Figure 4.2 shows the symbol selection process without pruning. The previous symbol table used for counting had only three symbols: “a”, “b”, and “c”. The frequencies in the table `count` of the used symbols for compression and their concatenations, in addition to their corresponding gains (`frequency` × `length`), were all computed by `compressCount`. We did not include double concatentations (`count3`) in this example for simplicity.

Assuming that the symbol table has a capacity of 5 symbols, `maketable` chooses the 5 symbols having the highest static gains, including the one-byte symbols “a” and “b”. Pruning those one-byte symbols and choosing “ba” and “ca” insetad would result in the best possible next symbol table, which will allow to compress the text in 5 bytes by using all two-byte symbols. Now we explain the reason of this suboptimal symbol selection and the suggested solution for it.

Problem. When choosing a symbol in `makeTable` that is formed by a concatenation of two or three symbols, we know that we overestimated the counts for the symbols that were used in the concatenation, especially if they are not yet chosen into the symbol table (i.e., they have less gain).

4.3.2 Pruning Logic

Count Correction. The idea of the count correction that was used is subtracting the count of the bigger symbol from the counts of all smaller symbols used in the concatenation and all successive pairs of them, in case the bigger symbol is a three-symbol concatenation. For example, consider the symbols A, B, C, and $D = ABC$ a concatenation of the three first symbols. When D is inserted in the new symbol table to construct, we reduce the count of D from the counts of the symbols A, B, C, AB, and BC. This reduction happens on the fly when choosing the symbols through the pop operations of the max heap. Then the updated symbols are pushed back to the heap with the updated gains and the old gains become invalid.

Negative Counts. This reduction may lead to some symbols having negative counts due to overlap, when the smaller symbol is part of different bigger symbols that are chosen in the symbol table. Those bigger symbols may overlap exactly at the small symbol causing a double subtraction from its count. A small example to illustrate this case is having a portion of a string compressed to symbols ABC, then choosing symbols AB and BC would decrease two occurences from the count of B, which actually appears only once. This problem can be modeled as an inclusion-exclusion problem, then the correction to the count of B from the example is to add the count of the intersection of AB and BC, which is ABC.

Including concatenations of symbols and raw bytes into the counts, and counting concatenation of symbols that may exceed the maximum symbol length make the simple addition of the count of the intersection symbol not enough to avoid negative counts. Therefore, the implementation deals with negative gains by simply discarding such symbols and not pushing them back to the heap. Empirically, we observed that discarding symbols whose gains become negative does not reduce the compression factor.

Other Pruning Direction. During symbol selection, if the smaller symbol occures before the bigger symbol that has it as a substring (the smaller symbol has more gain), it will not be pruned. Pruning such symbols proved to be significantly slower. A separate intermediate set has to be maintained for symbols that can be pruned, therefore not pushing them directly to the heap until the whole symbol selection process finishes. When pruned, they have to be removed from the intermediate set and pushed back to the heap.

As there was no noticeable benefit from this pruning direction in terms of compression factor, the symbol selection method pushes popped symbols from the max heap,

that have not been pruned, directly to the new symbol table. Pruning will consequently only affect symbols with less gain that will occur afterwards in the extraction order.

4.3.3 Implementation

```
1 def push_cand(code1: int, code2: int, code3: int, gain: int):
2   if gain <= 0:
3     return
4   heapq.heappush(heap, (-gain, code1, code2, code3))
```

Listing 4.5: Implementation of pushing candidates into the heap.

Heap Initialization. As the python implementation in Lst. 4.5 shows, the symbol to push is always considered as the concatenation of three symbols and their codes are pushed to the heap along with the gain. Here, the gain is negated because the python `heapq` module implements a min heap. If the symbol to push is not a concatenation or is a concatenation of two symbols, `code2` and `code3` are set to -1 denoting their absence. The pruned symbols with non-positive gains are discarded as discussed earlier.

The updated `makeTable` function now constructs the candidates from the `count1` and `count2` arrays as usual, but for three-symbol concatenations a loop over the HashMap of `count3` is added. Then the candidates are pushed with the `push_cand` helper function before entering the while loop to choose the symbols of the next symbol table.

```
# Fill a new symbol table
res = SymbolTable()
seen: set[bytes] = set() # avoid inserting duplicates
while res.nSymbols < MAX_REAL_SYMBOLS and heap:
  # g is negative gain
  g, code1, code2, code3 = heapq.heappop(heap)
  s = # reconstruct symbol
  curcnt = # restore current count
  if g != -curcnt * len(s) :
    continue # gain is invalid
  if s in seen:
    continue
  seen.add(s)
  res.insert(s)
  if code2 == -1:
    continue # nothing to prune
  # Pruning
  # prev is the previous symbol table
  L1 = len(prev.symbols[code1])
  L2 = len(prev.symbols[code2])
  if code3 == -1: # two-code symbol
    # Update the counts of prefix and suffix symbols
    count1[code1] -= curcnt
    if code1 != code2:
      push_cand(code1, -1, -1, L1 * count1[code1])
    count1[code2] -= curcnt
    push_cand(code2, -1, -1, L2 * count1[code2])
  else: # three-code symbol
    L3 = len(prev.symbols[code3])
    # Update the counts of one code symbols
    count1[code1] -= curcnt
    if code1 != code2 and code1 != code3:
      push_cand(code1, -1, -1, L1 * count1[code1])
    count1[code2] -= curcnt
    if code2 != code3:
      push_cand(code2, -1, -1, L2 * count1[code2])
    count1[code3] -= curcnt
    push_cand(code3, -1, -1, L3 * count1[code3])
    # pruning of two-code symbols
    L23 = min(MAX_SYMBOL_LEN, L2 + L3)
    count2[code1][code2] -= curcnt
    if code1 != code2 or code2 != code3:
      push_cand(code1, code2, -1, (L1 + L2) * count2[code1][code2])
    count2[code2][code3] -= curcnt
    push_cand(code2, code3, -1, L23 * count2[code2][code3])
```

Listing 4.6: Symbol pruning implementation.

Filling the New Table. This loop has now to first reconstruct the popped symbol and its current count from the codes, as shown in Lst. 4.6. Then if the symbol has been already pruned, it will be discarded. This check is performed through the comparison of the symbol's current gain with its gain at insertion time into the heap.

Next comes the insertion of the symbol to the new symbol table followed by the pruning logic. The `if` statements that precede some of the `push_cand` operations (lines 24, 32, 35, and 42 in Lst. 4.6) avoid pushing the same pruned symbol twice into the heap, as only the last push will be valid. This saves the insertion time and also the time for extracting those invalid states of the symbols. Therefore, if some symbol appears as part of the bigger symbol in multiple ways, it will be pushed only once to the heap after all reductions of its count.

Symbol pruning complements the two previously introduced contributions by partially reducing the conflicts between symbols and their concatenations obtained from `compressCount`, especially after adding a third counter. Furthermore, pruning helped introduce more symbols, that would have been ignored, giving more options for the DP function to find a better compression with the new set of symbols in the following generation.

5 Evaluation

In this section we will discuss the effects of the three introduced contributions on the original FSST algorithm in terms of compression factor and encoding speed across different datasets. The contributions will be also tested against DICT FSST [7], which combines dictionary encoding with applying FSST to the dictionary values.

5.1 Benchmarking Data

5.1.1 Dbtext

This is the dataset used for benchmarking in the original FSST paper [6]. It includes real-world text data from different sources. The data consists of 23 string columns ranging from 130 KB (city) to around 6 MB (urls) in size. They can be grouped in five categories based on their data nature:

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

5.1.2 NextiaJD

NextiaJD [10] has a collection of diverse string columns, dedicated to learning-based research. The selected tables from NextiaJD are:

- github-issues: it has the largest column of the considered benchmarking data in this thesis (body) with 32 MB size.
- glassdoor: the table with the most string columns, the majority being URLs and path links.
- Homo_sapiens.GRCh38.92: duplicate heavy data and identifiers.

- Parking_Violations_Issued_Fiscal_Year_2017: containing street names and columns with short rows.
- Reddit_Comments_7M_2019: the majority of columns are usernames, but also contains one text column (body) and one column with links (permalink).
- reviews_detailed: contains large text data (comments).

5.1.3 PublicBIbenchmark

PublicBIbenchmark is a dataset produced by CWI (Centrum Wiskunde & Informatica) with data representing real-world business intelligence workloads [11]. The data selected combine geographical locations, URLs with prefix-heavy patterns, and user-generated data in different languages containing special characters (hashtags) to test the effect of the improvements on heterogeneous data. The tables chosen from this dataset are "HashTags/Hashtags1" and "IGlocations2/IGlocations2_2". Each of these tables has a significant amount of string columns. Combined, they contain all mentioned types of data.

5.1.4 CyclicJoinBench

CyclicJoinBench is a dataset used to evaluate join algorithms on cyclic query graphs [12]. The datasets contains three identical tables regarding content types, therefore only one table was used for benchmarking (SNB1-parquet). Most of the string columns of this dataset have short rows featuring names, titles, and locations.

5.1.5 ClickBench

ClickBench is a dataset developed by ClickHouse [13], used to benchmark the performance of over 50 databases on simple analytical workloads. This makes the dataset suitable for the evaluation of compression methods. ClickBench has only three string-type columns that were all used for benchmarking: Title, Referer, and URL.

5.2 Benchmarking Method

A filtering process was applied on the selected tables from the datasets mentioned above, in order to get string columns suitable for benchmarking the effects of the improvements.

First, each table is reduced to its first 100 K rows. Then, to eliminate columns with numeric values, the only allowed column type was VARCHAR. Furthermore, an additional check was applied to filter out columns having date or datetime data.

To simulate real-world workload conditions, only columns having at least 1 K rows are kept for benchmarking.

Another filtering condition was setting an upper-bound of 35 MB for the columns. Larger columns have very long strings in each row and are usually compressed best using block-based compressors like zstd.

Dictionary Encoding. Duplicate-heavy columns in the datasets can be better compressed with dictionary encoding, which consists in keeping a dictionary with the unique values and a second field that contains, for each row in the original column, the index of the corresponding value in the deduplicated column. DICT FSST also uses dictionary encoding in its first phase, then applies FSST to compress the dictionary, which leads to higher compression factors.

The condition used for filtering is then: FSST compressed size $\leq$ Dictionary encoding size. The size of the dictionary is calculated as the sum of the lengths of unique strings and the total size of the integers used for mapping (bitpacked).

Columns that are filtered with this condition will be included again when benchmarking the effect of introducing the contributions to the second phase of DICT FSST.

5.3 Results

5.3.1 Ablation Study

We are going to discuss the effect of the contributions of this thesis (DP, third counter, and pruning) on the FSST algorithm regarding compression factor (CF) and encoding time. The resulting codebase [9] is an extension of the original FSST code [8], allowing the inclusion of the improvements with command line options as follows:

- no options: Exactly FSST will be run.
- dp-train: The DP function will be used for building the symbol table instead of the greedy `findLongestSymbol`.
- dp-encode: The DP function will be used for compressing the whole corpus after constructing the symbol table.
- triples: `compressCount` will use three counters instead of two.

- prune: Pruning will be used in the filling of the new symbol table in each generation in `makeTable`.

Running the basecode with any configuration different than FSST (i.e, with at least one of the options above) will discard some of the heuristics used by the original FSST code. These heuristics include a $\times 8$ boost to the gain of one-byte symbols, using fractions of the sample incrementally across generations instead of training on the whole sample for the 5 generations, and setting an entry threshold for the count of the symbols before pushing them to the heap of candidates.

The program running with all the above mentioned options will be called "BtrFSST", where all improvements are combined with FSST as a base.

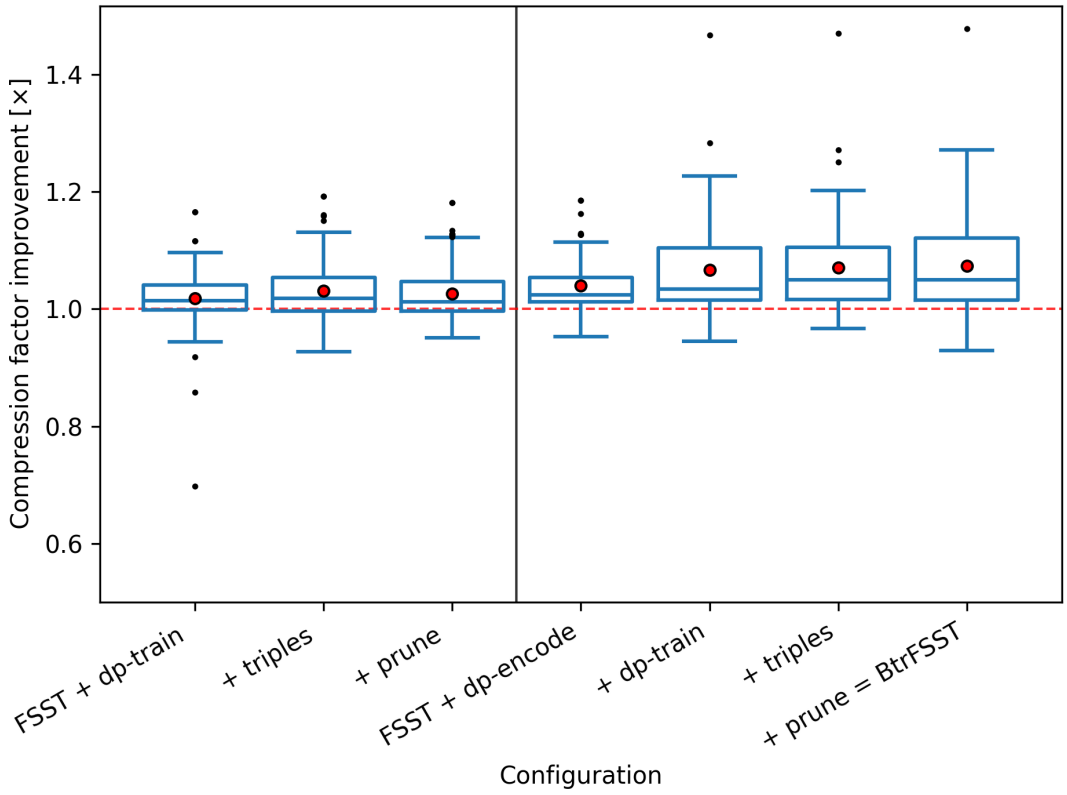


Figure 5.1: Improvement of the contributions on the FSST compression factor over all datasets combined.

CF Benchmarking. Figure 5.1 shows the compression factor improvement over all the selected columns of the datasets with different configurations. All improvements are

measured against FSST (red line). The left side adds the improvements, affecting only the symbol table construction (the full column compression is still greedy). The right side always includes DP in compression and then adds the contributions gradually in the same way.

The average improvement of BtrFSST was by 7.35%. Without including DP in compression, the contributions have an average improvement of 2.5% by solely improving the symbol table construction, while the improvement was by 3.4% when DP was used on compression. This shows that not using the greedy `findLongestSymbol` at all has a stronger impact on the overall compression factor. The implementation backs this claim, as when DP is used on both table construction and compression, there is no need to store symbols in a hash table. Therefore in that case, all symbols selected will be used in the symbol table, without the possibility for collisions, which was certain for symbols sharing the first three bytes, the key for the hash function in FSST.

The DP approach in the compression has the highest effect on the average compression factor improvement with 3.9% over FSST with greedy encoding. Then comes DP in table construction (2.6% with DP in compression and 1.7% without), which shows that the greedy approach was the main reason behind the sub-optimal compression factors of the original FSST. Although adding a third counter and using symbol pruning during symbol table filling did not have significant effects on the overall compression factor improvement, they can still be helpful for some workloads (discussed later).

Compression Speed Benchmarking. Figure 5.2 shows the encoding speed slowdown with the different configurations over all columns. The baseline is here again FSST (red line). The machine used for this benchmarking is a laptop equipped with an AMD Ryzen 9 CPU with 8 cores and 16 threads, operating at a base frequency of 3.3 GHz. The system uses 16 GB of RAM and an integrated AMD Radeon Graphics, with storage being provided by an NVMe SSD. All benchmarks were executed with the device connected to external power and configured in high-performance mode. No SIMD code was executed, meaning that the compression method used for original FSST was the scalar one, and not with AVX-512 which would be $\times 2.5$ faster [6].

The benchmarks show that BtrFSST is on average 78.3% slower than FSST. The added compression time overhead comes primarily from DP in encoding, which alone increased average compression speed by 63%. The contributions affecting the symbol table construction made the algorithm 30.8% slower than FSST, but added only a 15.3% slowdown comparing to FSST with DP in encoding. This proves that the bottleneck is the full column compression with DP, especially with columns that are significantly larger than the sample size (16 KB).

Worst Performance. BtrFSST performed worse than FSST on 10 files from a total of

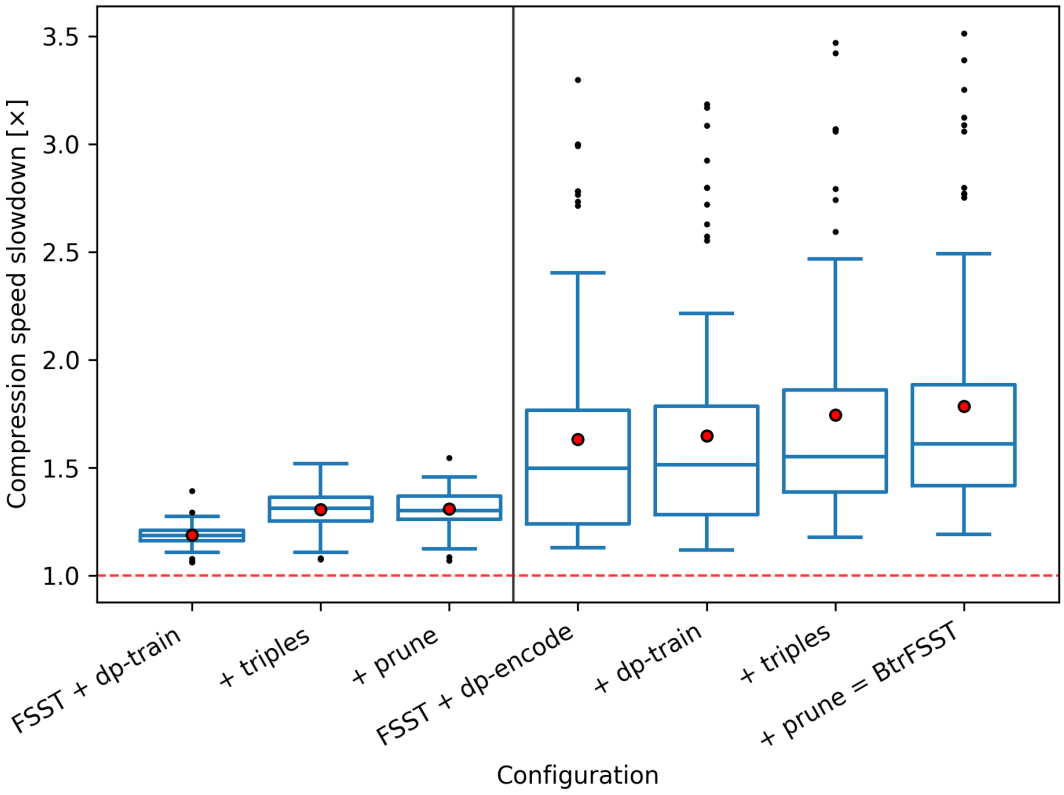


Figure 5.2: Effect of the contributions on the compression speed over all datasets combined.

92 files of the whole datasets combined. The column with the worst performance against FSST (7.6% less compression factor) is Reddit_Comments_7M_2019::id from the NextiaJD dataset, while all other columns are not more than 3.1% worse than FSST. For the id column the compression factor was the highest using only DP in table construction and encoding with a compression factor of 1.68 against 1.64 for FSST.

We can conclude that some of the contributions of this thesis may be degrading the compression factor for some columns. This issue will be addressed in future work to gurantee that no contribution has a negative effect on the compression factor, regardless of the sampled data and the type of data in the column.

Best Performance. BtrFSST performed significantly better than FSST on the majority of the columns, with 6 columns having more than 20% compression factor improvement. The column with the best performance, reaching an improvement of 47.7% in the compression factor, is Parking_Violations_Issued_Fiscal_Year_2017::Registration_State from the NextiaJD dataset. This column features state abbreviations, with each row having a string of length two. Experiments showed that BtrFSST performs best on columns with shorter strings and strings including numeric values.

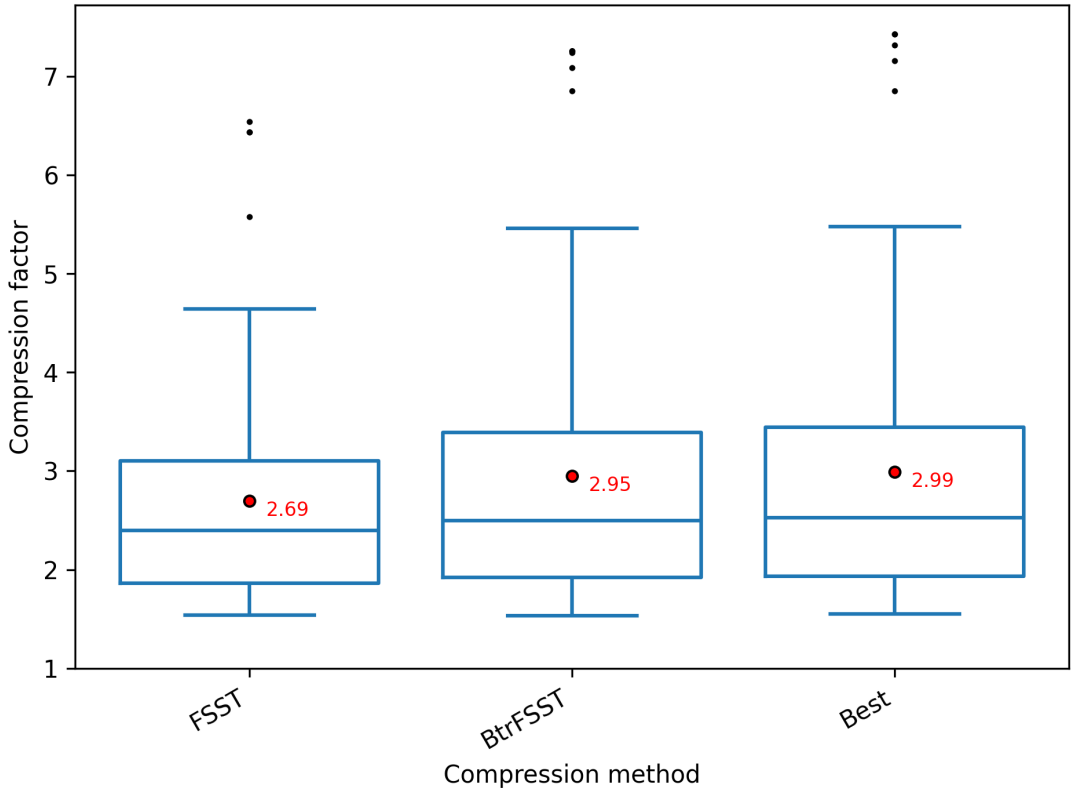


Figure 5.3: CF comparison when choosing the best configuration.

Best Configuration. Figure 5.3 shows the aggregate compression factors for choosing the best configuration for each file independently from the 8 configurations shown in Figure 5.1 (Best). The average compression factor improved from FSST by 11.1%, which is not significantly higher than always compressing with BtrFSST, that shows a 9.6% improvement in the average compression factor. Below are the counts for each configuration, of how many files it had the best compression factor:

- FSST: 2 (2.1%)
- FSST + dp-train: 5 (5.4%)
- FSST + dp-train + triples: 5 (5.4%)
- FSST + dp-train + triples + prune: 2 (2.1%)
- FSST + dp-encode: 6 (6.5%)
- FSST + dp-encode + dp-train: 18 (19.5%)
- FSST + dp-encode + dp-train + triples: 24 (26.0%)
- BtrFSST: 30 (32.6%)

The distribution proves that BtrFSST is the best configuration, although a noticeable number of files compress better with a subset of the improvement and their compression factors drop when adding other options. However, this drop is not significant and a rule of thumb can be using BtrFSST by default unless small improvements are worth the additional computational overhead of compressing the same file multiple times.

5.3.2 Block-Based Compressors

The two Figures 5.4 and 5.5 show that for the selected datasets, general purpose compressors perform better than both FSST and BtrFSST in terms of compression factor and compression speed. This is due to the high shared substrings between the rows of the datasets, especially prefixes in URLs and identifiers. LZ4 has a $\times 1.9$ better compression factor and is $\times 6.8$ faster than BtrFSST, while Zstd has a $\times 3.7$ higher compression factor and is $\times 4$ faster than BtrFSST. LZ4 is therefore optimized for speed while Zstd gives better compression factors which supports what was mentioned in Section 3.1.

Although both LZ4 and Zstd are faster and compress better than both BtrFSST and FSST, these block-based compressors stay inefficient for decompressing individual strings, as the whole surrounding block must be first decompressed before accessing a single string.

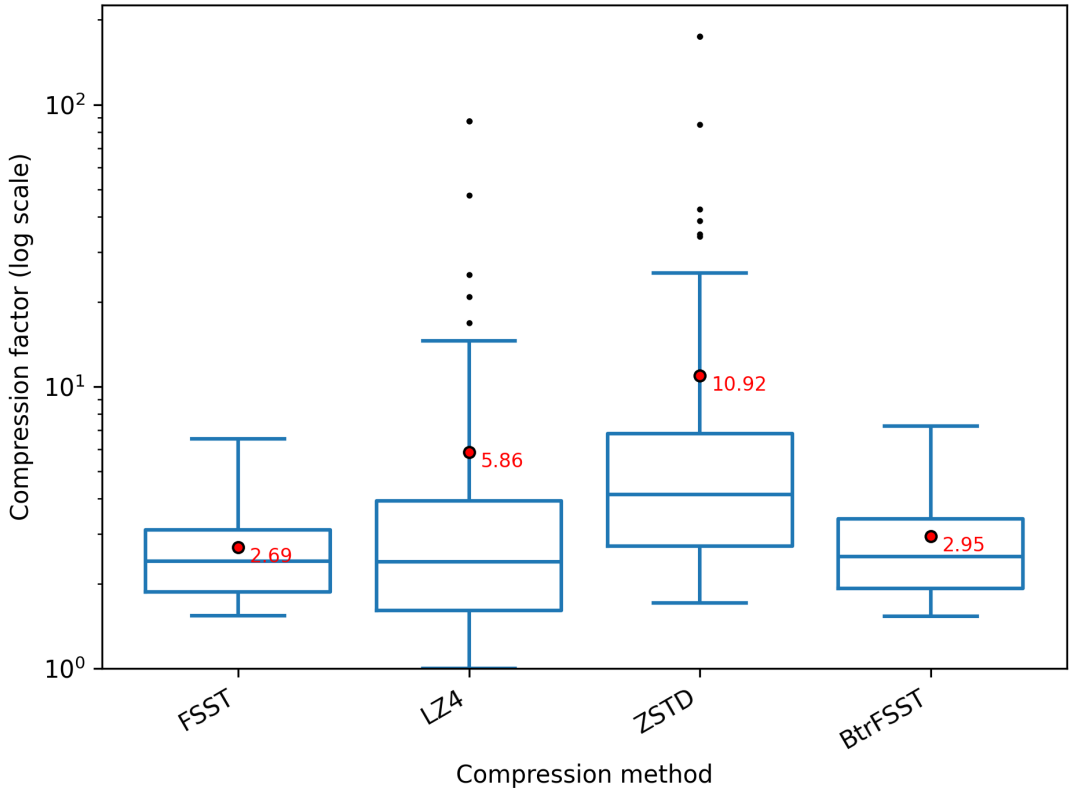


Figure 5.4: Comparison of compression factors with LZ4 and Zstd.

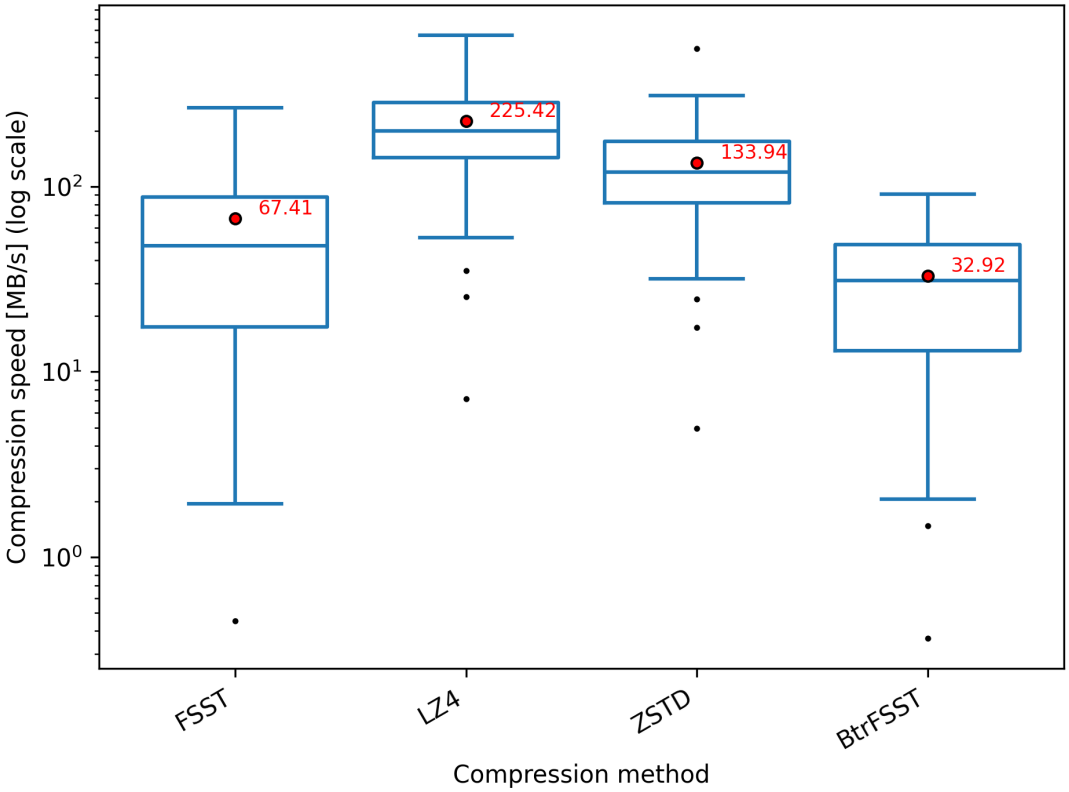


Figure 5.5: Comparison of compression speed with LZ4 and Zstd.

5.3.3 DICT FSST

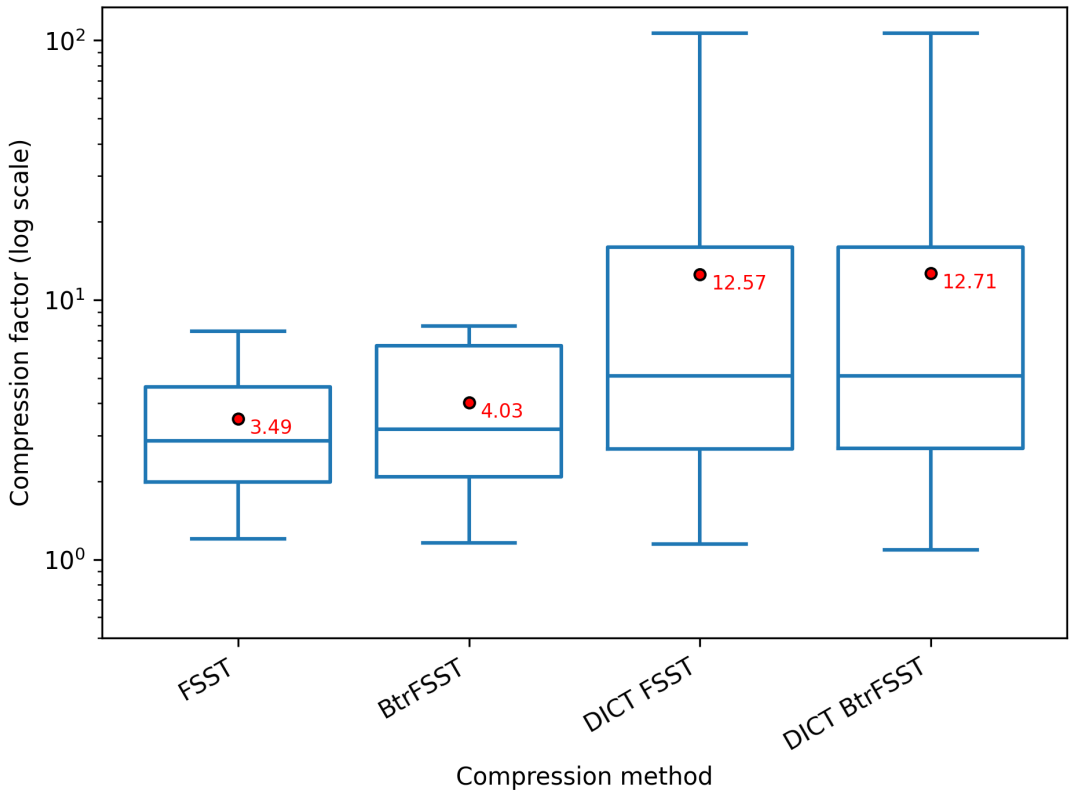


Figure 5.6: Effect of improvements on the compression factor with DICT FSST over all datasets combined.

Figure 5.6 shows the effect of the contributions on the DICT FSST compression. All columns filtered by the condition on the dictionary encoding size mentioned in Section 5.2 are included in this benchmark.

DICT FSST benefited from the duplicate-heavy columns, as it shows a higher compression factor average compared to FSST by $\times 3.6$.

BtrFSST performs better on such columns and shows a higher compression factor improvement against FSST than on the data without duplicate-heavy columns (15.4% to 9.6%). But when including dictionary encoding before BtrFSST (DICT BtrFSST), the effects of the contributions are no longer as significant. The small improvement of 1.1% is due to the smaller dictionaries, making the compression mainly influenced by the dictionary encoding phase.

6 Discussion & Future Work

Runtime Improvement. As discussed in the previous section, BtrFSST is almost twice as slow as the original FSST algorithm. This slowdown comes mainly from the DP in compression, which constitutes the bottleneck of the compression algorithm. Iterating over the whole corpus and computing the DP value at each position with at most 8 operations (for symbol search) introduced a significant overhead.

An alternative function that we tried during the thesis used a hybrid approach: when the last 8 DP values are close (MAX - MIN $\leq$ THRESHOLD), use the greedy `findLongestSymbol` and consider it the optimal symbol at the current position. Unfortunately, keeping the minimum and the maximum of the last seen DP values did not allow speeding up the function noticeably, although that was done in constant time by maintaining two `deques`. Furthermore, setting values for the threshold ≥ 2 dropped the CF significantly, rolling back most of the improvement achieved through the contributions.

Also avoiding branches in the symbol search did not bring any speedup, as the trie search should break with high probability after the first two to three operations, as most of the symbols have lengths of two and three bytes.

A possible area of improvement can be optimizing the symbol search with SIMD operations or better ways of storing the symbols. An immediate improvement can be also optimizing the heap structure used for storing candidate symbols in `makeTable`, as the pruning mechanism can make many symbols invalid and therefore wasting operations on extracting and discarding them.

Pruning Optimization. The pruning mechanism itself has a significant potential for improvement. The simple subtraction of counts of bigger symbols from their subsymbols needs to be further investigated. The inclusion-exclusion method discussed in Section 4.3 does not work currently because of the way the concatenations are made in `compressCount`. Therefore, considering another concatenation schema may allow for a perfect pruning logic which chooses the best matching symbols and reduces conflicts as much as possible given the set of candidate symbols.

The gain heuristic plays an important role, being the key for symbol evaluation and selection. Making the gain a dynamic function instead of a static one would remove the need for a pruning mechanism and improve the symbol selection process significantly.

Global vs. Local Optimizations. The improvements on symbol table construction, discussed in this thesis, provide better alternatives for symbol selection methods within each generation, but still relies on the same concatenation logic to improve the symbol table between generations. Adding a third counter helps the algorithm to converge faster by favoring longer symbols, but the converging path is still based on the same logic as the original FSST algorithm. These improvements are therefore mainly local and do not change the shape of the symbol table construction globally.

A straightforward approach for creating a symbol table with a single pass over the data was tried during the thesis. A Breadth First Search (BFS) algorithm was first employed. The sample data was modeled as a graph with each position as a node and the compression size as the distance to the last node that must be minimized. The state is then the position in the data and the current symbol table to build. Transitions can be made by using a symbol from the symbol table (matching a prefix), adding a symbol to the table and using it, or escaping the current byte. As escaping adds two bytes to the compressed size, an intermediate state was added to split the escaping step into two one-byte steps and be able to apply the usual BFS. This brute force idea is clearly unfeasible due to enormous memory requirements. We experimented with reducing the memory footprint by hashing the state and limiting the maximal number of symbols, but the compression factor suffered immediately after any reduction in the state.

This thesis kept therefore the core logic of FSST table construction and introduced improvements within its initial functions. Other attempts, that do not have to be just brute force, can be studied further to make global optimizations instead of or in addition to the learning process of the symbol table across multiple generations.

7 Conclusion

Reduced Greediness. The contributions discussed in this thesis helped with reducing the greedy aspect of the symbol table construction and compression in FSST. The DP approach that replaced the greedy function improved the quality of chosen symbols, optimizing each generation locally, which also affected the resulting symbol table. The DP approach proved to be more effective during final compression after building the symbol table. Replacing the greedy approach allowed smaller symbols to be chosen during `compressCount` which led to the second contribution (third frequency counter) to favor longer concatenations. Then symbol pruning was introduced to correct the gains of the candidate symbols on the fly during the table filling process of each generation. The gain heuristic is static and does not account for conflicts between overlapping symbols, which contributed to the overall greedy aspect of the original FSST algorithm.

Better Compression Factor. BtrFSST, the algorithm including all of the contributions, improved the average compression factor of the original FSST algorithm by 9.6%. This was the conclusion of benchmarking both methods over a wide selection of heterogeneous real-world string columns, including different types of information like URLs, dates, names, codes and long texts. The best compression factor improvement recorded in the datasets was 47.7%, while the average improvement was by 7.3%, which shows the potential of the contributions in compressing data exhibiting common substrings, especially digit-containing data. The extensive benchmarking of the different combinations of improvements showed a possibility to obtain a better compression factor when using only subsets of the contributions. However, this compression factor improvement is still not significant, as it requires multiple compressions for the same column only for an additional 1.4% average compression factor gain (over FSST).

BtrFSST also proved to be useful for other compression methods that use FSST like Dict FSST, which benefited with an average compression factor improvement of 1.1% with duplicate-heavy data. This improvement depends mainly on the amount of duplicates in the data: the fewer duplicates there are, the more the compression factor improves.

Runtime Overhead. When benchmarking the average runtime of both FSST and BtrFSST on the selected datasets, a slowdown of 78.3% was documented. This runtime

overhead comes primarily from adding DP to the full column compression after the symbol table construction. The improvements on the symbol table construction had a 30.8% slowdown, making encoding the real bottleneck of BtrFSST. Further research on the possibilities of reducing branching or using SIMD instructions in the DP function can reduce this overhead significantly.

BtrFSST still uses the same decompression method of FSST, whose speed is more decisive when it comes to using compressed data in database systems for query throughputs, as the data is only compressed once.

List of Figures

List of Listings

Bibliography

[1] A. van Renen, D. Horn, P. Pfeil *et al.*, 'Why tpc is not enough: An analysis of the amazon redshift fleet', *Proceedings of the VLDB Endowment*, vol. 17, no. 11, pp. 3694–3706, 2024. DOI: 10.14778/3681954.3682031.

[2] P. A. Boncz and M. L. Kersten, 'Monet: An impressionist sketch of an advanced database system', *Proc. IEEE BIWIT workshop, San Sebastian (Spain)*, pp. 240–251, 1995.

[3] D. J. Abadi, S. R. Madden and N. Hachem, 'Column-stores vs. row-stores', in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, L. V. S. Lakshmanan, R. T. Ng and D. Shasha, Eds., New York, NY, USA: ACM, 2008, pp. 967–980. DOI: 10.1145/1376616.1376712.

[4] *Facebook/zstd*, Original release: 2015-01-24. [Online]. Available: https://github.com/facebook/zstd (visited on 10/01/2026).

[5] Y. Collet, *lz4*, Original release: 2011-03-25. [Online]. Available: https://github.com/lz4/lz4 (visited on 10/01/2026).

[6] P. Boncz, T. Neumann and V. Leis, 'Fsst: Fast random access string compression', *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2649–2661, 2020. DOI: 10.14778/3407790.3407851.

[7] T. Bruineman, *[compression] introduce dict_fsst compression method*, Jan. 2025. [Online]. Available: https://github.com/duckdb/duckdb/pull/15637 (visited on 23/01/2026).

[8] P. Boncz, *fsst*, Original release: 2020-03-03. [Online]. Available: https://github.com/cwida/fsst (visited on 10/01/2026).

[9] H. Chehaidar, *btrfsst*, Original release: 2025-12-07. [Online]. Available: https://github.com/Hedi-Chehaidar/btrfsst (visited on 13/01/2026).

[10] dtim-upc/NextiaJD, *Nextiajd*, Original release: 2021-05-17. [Online]. Available: https://github.com/dtim-upc/NextiaJD (visited on 23/01/2026).

[11] B. Ghita, P. Boncz and D. Tomé. 'Public bi benchmark - part 1'. (Feb. 2019), [Online]. Available: https://zenodo.org/records/6277287 (visited on 23/01/2026).

[12] P. Groß, D. ten Wolde and P. Boncz, 'Adaptive factorization using linear-chained hash tables', 2019.

[13] ClickHouse/ClickBench, *Clickbench*, Original release: 2022-07-11. [Online]. Available: `https://github.com/ClickHouse/ClickBench` (visited on 23/01/2026).