Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master's Thesis

FSST+: Enhancing String Compression Through Common Prefix Extraction

Author: Yan Lanna Alexandre (2773394)

1st supervisor:	Prof. Dr. Peter Boncz
daily supervisor:	Paul Groß
2nd reader:	Pedro Holanda

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 5, 2025

Abstract

Modern analytical database systems process vast quantities of data, a significant portion of which is composed of strings. While general-purpose compressors like Zstandard achieve high compression ratios, their block-based nature impedes the fast random access required for efficient query processing. Conversely, lightweight schemes like the Fast Static Symbol Table (FSST) offer excellent random-access speed but fail to exploit longer-range redundancies, such as the shared prefixes common in URLs, file paths, and other structured identifiers.

This thesis introduces FSST+, an enhanced string compression system that extends the FSST algorithm to effectively compress prefix-heavy datasets while preserving its fast random-access capabilities. The core of our contribution is a novel compression layout that eliminates prefix redundancy by storing common prefixes only once per data block. Unique suffixes then reference these shared prefixes using lightweight offsets, enabling the direct and efficient decompression of any individual string without scanning neighboring data. To identify optimal prefixes, we developed and evaluated two strategies. Our most effective method employs a dynamic programming algorithm that operates on small, cache-friendly blocks of strings. By first sorting strings in runs of 128 elements, the algorithm finds an optimal partitioning into "similarity chunks" that maximizes the compression gains from prefix sharing. This approach is guided by a precise cost model and, by being constrained to fixed-size runs, maintains a practical linear-time complexity across the entire dataset. Our comprehensive evaluation on real-world analytical benchmarks demonstrates the efficacy of FSST+. On prefix-rich columns, FSST+ improves the compression ratio by an average of 70% compared to standard FSST. The most significant results were achieved by applying FSST+ to dictionary-compressed data. A DICT FSST+ implementation, particularly when combined with lexicographically sorting the dictionary, consistently achieved compression ratios superior to Zstandard on the tested datasets. This result is notable, as FSST+ provides a higher compression density than leading block-based methods while retaining the crucial advantage of fast random access, positioning it as a powerful and practical addition to the compression toolkit for modern analytical systems.

Contents

List of Figures

LIST OF FIGURES

List of Tables

List of Listings

1

Introduction

Recent research in cloud analytics and benchmarking [17; 19] has shown that a significant portion (around 50%) of tabular data in modern analytical systems is composed of strings.

In light of these findings, there is a clear need for advanced string compression techniques that not only achieve high compression ratios but also support fast random access, allowing individual strings to be decompressed directly without requiring the processing of entire blocks.

Textual data often lacks a rigid structure; it can consist of a wide range of byte values and exhibit significant variations in length. However, among commonly stored string columns, patterns can be found. One particular pattern that can be observed in many dataset columns is the presence of URLs. In fact, according to a recent study analyzing various CSVs on GitHub, URLs appear [14] in the top 10 most common types of data stored. In the data stored by URL columns, there is often a shared prefix followed by changing suffixes. For example, a set of URLs might appear as:

`https://www.reddit.com/r/technology`
`https://www.reddit.com/r/dataengineering`
`https://www.reddit.com/r/pics`

The redundant storage of the common prefix, `https://www.reddit.com/r/`, represents a clear opportunity for more effective compression. However, existing random-access compressors, such as the Fast Static Symbol Table (FSST) algorithm, are highly optimized for decompressing individual strings based on substring-level symbol substitution but do not inherently exploit these longer-range prefix redundancies that span across multiple strings.

This observation forms the central motivation for this thesis: to develop an enhanced compression system that extends the FSST algorithm to specifically target these shared prefixes. We introduce **FSST+**, a novel approach designed to eliminate prefix redundancy by storing common prefixes only once for a group of strings, thereby improving compression ratios while rigorously preserving the fast random-access capabilities critical for analytical database performance. This endeavor leads us to the following research questions:

RQ1: To what extent do common prefixes occur in the string columns of the ClickBench[6], NextiaJD[8], CyclicJoinBench[12], and Public BI Benchmark datasets [11], and how can the potential for prefix-based compression be quantified?

RQ2: How can a random-access string compression scheme like FSST be extended to efficiently compress shared prefixes, and what data structures are required to support this functionality without sacrificing its core benefits of fast, individual string decompression?

RQ3: What algorithmic approaches can effectively and efficiently identify optimal prefixes within a corpus to maximize the compression ratio?

RQ4: How does the proposed FSST+ scheme compare against established baseline compressors (FSST, Dictionary Encoding) and leading block-based compressors (Zstandard, LZ4) in terms of compression ratio and compression speed on representative analytical workloads?

1.1 Contributions

Our main contributions are as follows:

- **Empirical Analysis of Prefix Prevalence:** We conducted a comprehensive empirical analysis of string columns derived from widely-used analytical benchmarks, specifically ClickBench [6], NextiaJD [8], CyclicJoinBench[12], and the Public BI Benchmark [11]. This detailed study quantifies the prevalence and characteristics of shared prefixes within real-world tabular datasets, identifying specific patterns (such as URLs and common text fragments) where prefix-based compression offers substantial opportunities for storage reduction. This analysis serves as the foundational empirical justification for the development of our proposed system.

- **FSST+ Design and Optimal Prefix Identification:** We introduce a novel random-access storage format designed to eliminate prefix redundancy by storing common prefixes only once for a group of strings. This design employs ”jump-back” offsets, allowing suffixes to efficiently reference their corresponding prefixes, thereby significantly reducing storage requirements for prefix-heavy string datasets while rigorously preserving fast, individual string retrieval. Alongside this format, we developed and implemented a dynamic programming-based algorithm that optimally identifies and matches the most advantageous prefixes to their corresponding suffixes, thereby maximizing the overall compression ratio.
- **Comprehensive Compression Performance Evaluation:** We conduct an extensive compression performance evaluation of the proposed FSST+ scheme against a diverse set of established baseline and state-of-the-art compression algorithms. This evaluation quantifies FSST+’s effectiveness across various real-world analytical workloads, assessing its compression ratios and compression speeds in comparison to Fast Static Symbol Table (FSST), Dictionary Encoding, Zstandard, and LZ4. Our analysis demonstrates the specific scenarios where FSST+ provides substantial gains in storage efficiency while maintaining its core advantage of fast random access, critically evaluating its trade-offs in practical application.
- **DICT FSST+ Hybrid Compression:** We introduce and thoroughly evaluate DICT FSST+, a novel hybrid compression scheme that combines the benefits of traditional Dictionary Encoding with the FSST+ algorithm. We demonstrate that by leveraging the inherent flexibility of dictionary order, lexicographically sorting dictionary entries before compression significantly enhances the efficacy of our prefix-sharing approach, establishing a practical upper bound on its compression potential. Our comprehensive evaluations demonstrate that on prefix-rich data, DICT FSST+ not only significantly outperforms standard DICT FSST but can also surpass the compression ratios of leading block-based compressors, such as Zstandard, while preserving the crucial attribute of random access.

1.2 Outline

The remainder of this thesis is structured to systematically address our research questions.

Chapter 2 provides the necessary context by reviewing fundamental and state-of-the-art string compression techniques. We begin with an overview of Dictionary Encoding

and the Fast Static Symbol Table (FSST) algorithm, which form the foundation of our work. We then examine general-purpose block-based compressors, such as Zstandard and LZ4, highlighting their strengths and limitations in terms of random access. The chapter concludes by surveying related academic research in specialized Dictionary Encoding and order-preserving schemes, establishing the landscape into which FSST+ is introduced.

Chapter 3 introduces the core technical contribution of this work: the FSST+ compression layout. We detail the design of this novel storage format, explaining its batch-based design, the structure of its global and block-level headers, and the mechanism for separating shared prefixes from unique suffixes. This chapter illustrates how the design eliminates prefix redundancy while preserving the fast, individual string random access that is critical for analytical database systems.

Chapter 4 details our benchmarking methodology. We describe the selected real-world datasets, including ClickBench, NextiaJD, CyclicJoinBench, and the Public BI Benchmark, chosen for their representative string-heavy workloads. We also explain the process of creating de-duplicated and sorted de-duplicated variants of these datasets to rigorously evaluate the effectiveness of prefix-sharing separately from simple duplicate elimination.

Chapter 5 presents our primary algorithm for identifying optimal prefixes. We introduce the concept of ”Similarity Chunks” and detail the dynamic programming solution used to find the most efficient partitioning of strings within a block. This chapter presents a formal definition of the algorithm and analyzes its linear-time complexity, demonstrating its optimality within block constraints. We provide a comprehensive evaluation of compression performance, including experimental results on parameter tuning; different blocksizes; and recursive applications of FSST and comparing it to FSST+ performance.

Chapter 6 explores an alternative heuristic-based approach. We describe a hash-grouping algorithm that uses hashing and a recursive splitting strategy to group strings with common prefixes without a preliminary sorting step. We analyze its implementation and present results that compare its efficacy in terms of compression ratio and speed against the dynamic programming solution.

Chapter 7 discusses the broader implications of our findings and outlines promising directions for future research, which include potential improvements to the dynamic programming algorithm, such as advanced Range Minimum Query optimizations, the potential for parallel execution, and a deeper analysis of decompression performance.

Finally, Chapter 8 concludes the thesis by summarizing our contributions and revisiting the research questions. We synthesize the results to provide a definitive answer on the

effectiveness of FSST+ as a high-performance, random-access string compression technique for modern analytical workloads.

2

Background and Related Work

To contextualize the contributions of this thesis, this chapter provides an overview of the relevant string compression techniques that form the foundation of our work. We begin by examining two fundamental approaches prevalent in analytical databases: Dictionary Encoding and the Fast Static Symbol Table (FSST) algorithm. Dictionary Encoding serves as a crucial baseline, representing the industry standard for handling low-cardinality string columns.

A detailed exposition of FSST is then presented, as its design principles for fast, random-access decompression directly inspire our proposed extensions. Finally, we situate these methods within the broader landscape by considering general-purpose block compressors, such as Zstandard and LZ4. While these algorithms often yield higher compression ratios, their block-oriented nature presents inherent limitations for workloads that require the rapid retrieval of individual data entries.

By analyzing the respective strengths and weaknesses of these established methods, we identify the specific opportunity that our research addresses: the efficient compression of shared prefixes in string data. This common pattern remains underexploited by existing random-access schemes.

2.1 Dictionary Encoding

Dictionary Encoding represents a fundamental approach to reducing storage requirements for columns containing repeated values. The technique works by identifying all unique values within a column, storing them once in a separate dictionary structure, and replacing the original data with integer codes that reference entries in this dictionary [1].

This approach proves particularly effective for unique value columns, such as categorical data, status codes, or product categories. For example, a column containing millions of rows with only hundreds of distinct values can achieve substantial compression ratios. The storage savings come from two sources: elimination of duplicate string storage and the compact representation of integer codes compared to variable-length strings. One way to make these integer codes compact is by bit-packing them. Bit-packing is a technique that stores values using only the minimum number of bits needed, reducing space by avoiding fixed-size representations. If we take U = number of unique values, then the minimum number of **bits** required to store a code is given by $\lceil \log_2 U \rceil$. These codes are then tightly packed into a byte stream, potentially crossing byte boundaries to avoid padding and maximize space efficiency.

Dictionary Encoding also provides significant query performance benefits beyond storage reduction. Integer comparisons and operations are substantially faster than string operations, enabling more efficient filtering, grouping, and join operations. Additionally, dictionary-compressed columns can leverage specialized algorithms designed for integer data, such as bit-packing techniques that further reduce storage requirements.

However, Dictionary Encoding introduces certain trade-offs. The approach requires maintaining the dictionary structure alongside the compressed data, which adds overhead for columns with low repetition rates. Additionally, operations that require access to the original string values must perform dictionary lookups, which can potentially introduce computational overhead. The effectiveness of Dictionary Encoding decreases as the number of unique values approaches the total number of rows, eventually becoming counterproductive when storage overhead exceeds compression benefits.

The technique integrates naturally with columnar storage systems common in analytical databases, where entire columns can be dictionary-compressed independently. Modern implementations often employ adaptive strategies that automatically determine whether Dictionary Encoding is beneficial for specific columns based on sample compression for cost estimation.

2.2 FSST

Fast Static Symbol Table (FSST) [4] is a lightweight string compression scheme designed for fast random access. Unlike traditional compression methods that operate on large blocks, FSST is optimized for compressing individual strings, making it particularly suited for database applications where random access to string data is crucial.

2.2.1 Core Concepts

The core idea behind FSST is to replace frequently occurring substrings (called ”symbols”) with shorter, single-byte codes. These symbols have a length between 1 and 8 bytes and are aligned on byte boundaries (not bit boundaries), which simplifies processing and improves speed. A key feature of FSST is its use of a **static** symbol table. This table, which maps symbols to codes, is constructed once and remains immutable during both compression and decompression. Here follows an example of how strings would be compressed by FSST:

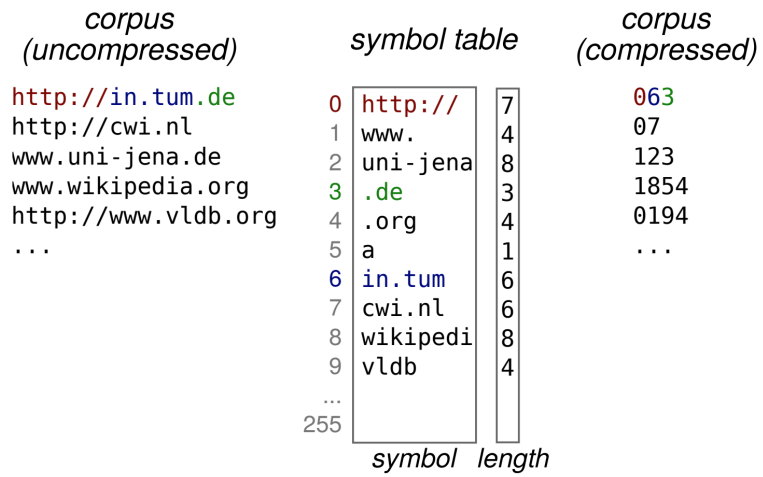


Figure 2.1: FSST Compression Visualization

2.2.1.1 Symbol Length and Encoding

FSST processes data in whole bytes. Each symbol is replaced by a 1-byte code, allowing for a maximum of $2^8 = 256$ distinct symbols (0 to 255, with symbol 255 being always reserved as an escape symbol) to handle bytes for which no symbol is found. This byte-level processing contributes to the algorithm's efficiency.

2.2.1.2 Static Symbol Table

The **static** nature of the symbol table is crucial for FSST's random access capability. Unlike methods like LZ4 [2], which maintain a mutable state during compression and decompression, FSST's fixed table allows any compressed string to be decompressed independently, without requiring the processing of preceding data. This contrasts sharply with block-based compression schemes, which require decompressing an entire block to access a single string.

2.2.2 Symbol Table Construction

The efficacy of FSST is fundamentally dependent on the quality of its static symbol table, which must be tailored to the patterns of the specific data it is intended to compress. The construction of this table is a greedy, iterative process that approximates a more computationally intensive optimal solution. This process operates not on the entire dataset, but on a representative, stratified sample of the data to ensure that the resulting symbol table generalizes well while keeping the construction phase computationally tractable.

The construction algorithm can be conceptualized as an evolutionary process unfolding over several generations. It begins with an initial set of symbols corresponding to the 256 possible single-byte values. In each subsequent iteration, or ”generation”, the algorithm seeks to improve the symbol table by creating longer, more valuable symbols. First, the current data sample is encoded using the symbol table from the previous generation. The algorithm then analyzes this encoded output to determine the frequency of all adjacent two-symbol pairs. For each observed pair, a potential new, concatenated symbol is considered. The ”value” of such a new symbol is calculated as the product of its length and its frequency of occurrence. This metric prioritizes symbols that are not only frequent but also offer significant length savings.

The symbol table for the next generation is then assembled by selecting the most valuable symbols. This selection includes the highest-value symbols from the existing generation, supplemented by the most valuable new symbols formed from concatenation. This ensures that effective symbols are retained while new, even more effective ones are introduced. The total number of symbols is constrained to 255, as one code is reserved for the escape mechanism. This iterative refinement process is repeated for five generations, by which point the symbol table typically reaches high quality. This construction method bears resemblance to the RePair algorithm [16], which also recursively replaces frequent pairs. However, while RePair replaces the single most frequent pair globally in each step, FSST evaluates all adjacent symbol pairs in the sample and generates a new symbol set in parallel during each iteration, making its construction process significantly faster.

On a side note, FSST also provides an option to generate 0-terminated strings for compatibility with C-style strings. In this mode, code 0 encodes the null byte (`'\0'`). This reduces the number of available codes to 254, slightly impacting the compression ratio, but enhancing interoperability.

2.2.3 Compression

Once the static symbol table is constructed, the compression process must rapidly identify the longest matching symbol at any given position in the input string. A naive search would be far too slow for a high-performance compressor. FSST achieves its high compression throughput by employing an optimized, branchless lookup strategy built upon two key data structures: a lossy perfect hash table for symbols of length three or more, and a comprehensive 2D array for shorter symbols.

The primary lookup mechanism is a lossy perfect hash table, which we will refer to as `hashTab[]`. The key for this table is derived from the first three bytes of the input string using a single multiplicative hash function. The term ”lossy perfect” is critical: during table construction, if two different symbols hash to the same bucket, one of them (the one with lower gain) is discarded. This ensures that at runtime, each bucket in the hash table contains at most one symbol candidate, thereby eliminating the need for collision resolution schemes such as chaining or probing. This design guarantees that a lookup requires only a single memory access to fetch a candidate symbol.

The secondary structure, `shortCodes[][]`, is an array of $256 \times 256 = 65536$, indexed by the first two bytes of the input. It is populated first with all 2-byte symbols from the symbol table. Afterwards, any remaining empty slot `shortCodes[A][B]` is filled with the code for the 1-byte symbol ‘`A`‘. Consequently, a lookup in this array using the next two bytes of input will always yield the code for the longest matching symbol of length one or two.

The compression kernel, detailed in Algorithm 4 of the original paper [4], integrates these two structures into a single, branch-free operation. For a given input position, the algorithm performs two lookups in parallel. It uses the first three bytes to probe the `hashTab[]`, retrieving a single candidate symbol for a potential long match. Simultaneously, it uses the first two bytes to find the guaranteed 1-or-2-byte match from the `shortCodes[][]` array. The algorithm then performs a single comparison to validate the candidate from `hashTab[]` against the input string. A conditional move instruction, which maps efficiently to modern CPU architectures and SIMD (Single Instruction Multiple Data) instruction sets, is then used to select the final code. If the `hashTab[]` lookup yielded a valid match, its code is chosen; otherwise, the code from the `shortCodes[][]` array is used. This design avoids loops and branches, enabling high compression speeds. If no symbol of length one or greater is found (which can only happen if a byte was not in the training sample and was not included as a default 1-byte symbol), the escape mechanism is used.

2.2.4 Decompression

FSST decompression is remarkably simple and fast due to the static nature of the symbol table. The core operation is a direct index lookup of the 1-byte code in the symbol table to retrieve the corresponding symbol.

This C++ code uses two arrays, `sym` (containing 8-byte representations of symbols) and `len` (containing their lengths). It then uses each code to do a symbol lookup and writes the symbol to the output.

To handle bytes not present in the symbol table, FSST uses code 255 as an escape marker. When encountered, the next byte in the input is copied directly to the output. This enables the compression of unseen data and allows for the construction of a symbol table from a sample. Algorithm 2.1 shows the C++ implementation, including escape handling.

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

Listing 2.1: FSST Decompression with Escaping (from original paper)

2.2.5 Summary

In summary, FSST presents a compelling trade-off between compression speed and ratio, specifically optimized for workloads requiring fast random access to individual strings. Its key strengths are the static, pre-computed symbol table and the efficient lookup-based decompression mechanism. These features enable the decoding of any single compressed string without requiring the processing of its neighbors.

2.3 DICT FSST

DICT FSST (Dictionary FSST) represents a hybrid compression technique that combines the benefits of Dictionary Encoding with the string-level compression capabilities of FSST. This approach was implemented and integrated into DuckDB as part of ongoing efforts to optimize string storage and processing performance [5].

The algorithm operates in two distinct phases. First, it applies traditional Dictionary Encoding to identify and de-duplicate all unique strings within a column, storing each distinct value only once in a dictionary structure. This initial step eliminates the storage overhead associated with repeated identical strings, which can be substantial in real-world datasets containing categorical data, identifiers, or standardized text fields.

The second phase applies FSST compression to the dictionary entries themselves. Rather than storing the unique strings in their original form within the dictionary, each string is compressed using FSST's symbol table approach. This dual-level compression strategy addresses different types of redundancy: Dictionary Encoding eliminates exact duplicates across the dataset, while FSST compression exploits substring patterns within the unique strings themselves.

DICT FSST is highly relevant to this thesis, as we can experiment with a DICT FSST+ version, substituting the old FSST algorithm with FSST+ for compressing the dictionary corpus, and then measure its performance.

2.4 LZ77

LZ77 is a foundational algorithm for lossless data compression, introduced by Abraham Lempel and Jacob Ziv in 1977. It works by scanning the input stream to detect sequences of bytes that have already appeared. To save on storage, LZ77 encodes them as a pair *(offset, length)* referencing where the sequence first appeared and how many bytes are repeated. This method is called a *dictionary-based* approach because the previously seen data acts like an evolving "dictionary" of possible matches.

In more concrete terms, suppose the compression algorithm encounters a sequence of bytes that matches a segment it processed just a few characters earlier. Rather than writing the sequence in full, it simply says, "go back 10 bytes and copy 5 bytes from there." This significantly reduces the total amount of data stored when repeated patterns occur.

2.5 Zstandard

Zstandard (often shortened to `ZSTD`) is a fast, lossless compression algorithm developed by Facebook and widely used across the industry. It aims to combine high compression ratios with high decompression speeds, making it suitable for a wide range of data-intensive applications, such as database storage, file systems, and network transmissions.

Compressor name	Ratio	Compression	Decompress.
zstd 1.5.6 -1	2.887	510 MB/s	1580 MB/s
zlib 1.2.11 -1	2.743	95 MB/s	400 MB/s
brotli 1.0.9 -0	2.702	395 MB/s	430 MB/s
zstd 1.5.6 –fast=1	2.437	545 MB/s	1890 MB/s
zstd 1.5.6 –fast=3	2.239	650 MB/s	2000 MB/s
quicklz 1.5.0 -1	2.238	525 MB/s	750 MB/s
lzo1x 2.10 -1	2.106	650 MB/s	825 MB/s
lz4 1.9.4	2.101	700 MB/s	4000 MB/s
lzf 3.6 -1	2.077	420 MB/s	830 MB/s
snappy 1.1.9	2.073	530 MB/s	1660 MB/s

Table 2.1: Performance comparison of various compression libraries on the Silesia compression corpus. [18][9]

We can see that, compared to the other methods in the benchmark, ZSTD provides the best compression ratio with great compression and decompression speed.

2.5.1 Block-Based Compression

ZSTD processes input data in chunks called *blocks*, typically up to 128 KB in size. Each block is compressed independently, allowing for localized optimizations. This design also facilitates the distribution of work across multiple threads, as different blocks can be processed in parallel. Despite being compressed independently, blocks can maintain a shared history buffer, allowing the compressor to reference recent data if the format dictates a window that spans multiple blocks. This approach strikes a balance between memory efficiency, flexibility in tuning compression levels, and the ability to handle large datasets.

2.5.2 LZ77-Style Match-Finding

Within each block, ZSTD uses an LZ77-like strategy (explained earlier in section Section 2.4) to locate repeated substrings. The algorithm scans for sequences that have appeared earlier in the same block or in the accessible history window. Whenever a repetition is detected, it is represented by an *(offset, length)* pair instead of storing the repeated text again. Data

segments that do not match previous text are treated as *literals*. By replacing lengthy repeats with offset-length references, ZSTD reduces the amount of literal data needing subsequent entropy coding, thus improving both speed and compression ratio.

2.5.3 Entropy Coding with FSE and Huffman

After the LZ77-based match-finding step identifies offsets, lengths, and literals, ZSTD applies entropy coding to compress these elements into a compact bitstream. The central technique used here is *Finite State Entropy* (FSE), supplemented in some configurations by Huffman coding. FSE is a variant of Asymmetric Numeral Systems (ANS) designed to provide high-speed decoding and efficient compression of discrete symbol distributions. By assigning shorter codes to more frequent symbols, ZSTD reduces the average number of bits used per symbol, thereby improving overall compression efficiency.

The compression process begins by gathering three types of information from each block: (1) the stream of *literal bytes*, (2) the *match length* codes, and (3) the *offset* codes. ZSTD constructs probability distributions for each of these symbol streams and then builds corresponding entropy tables. If FSE is used, each distribution is represented as a set of states in an ANS-based table. When Huffman coding is chosen (often for simpler or less skewed distributions), a binary tree is built instead. Once these tables are established, the sequences of match lengths, offsets, and literals are encoded accordingly:

- In **FSE mode**, each symbol (e.g., a match length) transitions the encoder or decoder from one state to another, using a table-based finite-state machine. This allows multiple symbols to be packed efficiently into fewer bits, leveraging the varying probabilities of different symbols.
- In **Huffman mode**, each symbol is mapped to a bit pattern determined by the Huffman binary tree; shorter bit patterns correspond to symbols with higher frequency. [13]

The outcome of this procedure is a single compressed bitstream for each block, containing all the required information to reconstruct offsets, match lengths, and literal bytes. During decompression, ZSTD initializes its decoder with the same entropy tables and reverses the process. For blocks encoded with FSE, the decoder tracks states by referencing the same state-transition tables, swiftly retrieving symbols from the compressed stream without extensive overhead. In Huffman-coded blocks, the bit patterns are traversed in the Huffman tree to decode symbols in a straightforward manner.

One of the defining advantages of FSE over Huffman is that FSE remains *stream-friendly*. With FSE, each symbol is processed by updating a single evolving state variable, so the decoder never needs to pause and wait for additional bits when a packet ends. This contrasts with Huffman decoding, where the binary tree traversal can be interrupted at a packet boundary, requiring extra buffering or state management. This efficient, continuous processing yields high throughput and fast decompression speeds that often exceed those of more traditional methods. Combined with LZ77-based match elimination, this fast entropy coding stage enables ZSTD to achieve a compelling balance between compression ratio and performance, making it ideal for both large-scale data processing and latency-sensitive applications.

2.5.4 Small Data Compression

ZSTD supports external dictionaries to enhance compression for data with highly repetitive or domain-specific patterns. This is a different type of dictionary from Dictionary Encoding. These dictionaries are usually created by analyzing representative samples of the target dataset, extracting the most frequent substrings, and storing them for future reference. This is done by using ZSTD's "training mode", which can be used to tune the algorithm for a selected type of data. The result of this training is stored in a file called "dictionary", which must be loaded before compression and decompression can occur. Using this dictionary, the compression ratio achievable on small data improves dramatically.

2.5.5 Implementation Details

The ZSTD implementation enables users to switch between different parsing strategies, allowing for a desired trade-off between compression ratio and speed. Greedy parsing offers faster performance, whereas optimal parsing seeks to find longer or more frequent matches at the cost of additional computation. Special block types, such as *uncompressed* blocks, handle edge cases or data that may not benefit from compression. During decompression, ZSTD strictly follows the encoded references (offsets and lengths) and literal streams to reconstruct the original data. This ensures that while the compressor can use complex parsing strategies, the decompression path remains consistently fast and deterministic.

2.5.6 Rationale for Zstandard's Design

By combining a block-based approach with LZ77-style parsing and efficient entropy coding, ZSTD provides both high compression ratios and rapid decompression. The block-based

structure simplifies parallelization, as each block can be independently compressed or decompressed. Meanwhile, the offset-length references drastically cut down on redundant data, leaving only a smaller stream of unmatched literals and match metadata for FSE or Huffman encoding. Overall, the algorithm's design is a balanced solution, offering adjustable trade-offs between speed and compression ratio, and it is especially effective for modern, large-scale data workloads.

2.6 LZ4

LZ4 is a lossless data compression algorithm known for its focus on speed, particularly in decompression [7]. It is a member of the Lempel-Ziv 77 (LZ77) family of compressors, which operate by identifying and eliminating duplicate data sequences within an input stream. The primary design goal of LZ4 is to prioritize compression and decompression throughput over achieving the highest possible compression ratio. Nevertheless, unlike schemes designed for fine-grained random access, such as FSST, LZ4 is fundamentally a block-based compressor.

The core mechanism of LZ4 involves partitioning the input data into sequences of literals and matches. Literals are segments of data that have not been seen before and are stored exactly as they appear. Matches, on the other hand, are sequences that have already appeared in the recently processed input, which acts as a sliding window dictionary. Instead of storing the repeated sequence, LZ4 encodes it as a reference consisting of an *offset* and a *match length*. The offset specifies how far back in the output stream to look for the start of the sequence, while the match length indicates how many bytes to copy.

What distinguishes LZ4 and enables its remarkable speed is the simplicity of its encoding format and the deliberate omission of a computationally expensive entropy coding stage. During compression, LZ4 employs a fast hash table to quickly identify potential matches for the current input position. Once a match is found, the algorithm greedily accepts it and continues processing, rather than spending additional cycles searching for a theoretically optimal match that might yield a slightly better compression ratio. This greedy parsing strategy minimizes CPU usage during the compression process.

The LZ4 stream format is designed for straightforward and rapid parsing during decompression. The data is structured as a series of blocks, where each block begins with a token. This token, a single byte, efficiently encodes the length of the subsequent literal sequence and the length of the following match. By reading this token, the decompressor

immediately knows how many literal bytes to copy directly from the compressed stream and how many bytes to copy from a previous location in the output buffer.

Crucially, LZ4 does not apply any form of entropy coding, such as Huffman coding or Finite State Entropy, to the literals or the match references. The literal bytes are written directly to the compressed output, and the offset-length pairs are encoded using simple, byte-aligned variable-length integers. While this means that statistical redundancies in the literal stream and match metadata are not exploited, it allows the decompressor to operate with minimal computational overhead. The decompression process consists almost entirely of memory copy operations (‘memcpy‘), which are highly optimized on modern CPU architectures. Consequently, LZ4’s decompression speed is often limited only by memory bandwidth, enabling throughputs of several gigabytes per second on a single CPU core.

In summary, LZ4 is a highly effective block compression algorithm that prioritizes throughput by using a simple, greedy parsing strategy and a state-dependent encoding format. Its design is predicated on processing contiguous blocks of data where the entire block is compressed and decompressed as a single unit. This architectural choice makes it unsuitable for applications requiring random access to individual elements within a compressed collection, a use case for which algorithms like FSST are specifically designed. Instead, LZ4 excels in system-level tasks where opaque blocks of data must be processed with small CPU overhead.

2.7 Practical string dictionary compression using string dictionary encoding

While the previous compression techniques focus on compressing string content directly, an orthogonal approach involves compressing the dictionary data structures that store and manage string collections. String dictionaries are fundamental components in database systems that map strings to unique identifiers, enabling efficient string processing and indexing.

Kanda et al. [15] address the challenge of compressing string dictionaries themselves, particularly for very large datasets where dictionary size becomes a bottleneck.

2.7.1 Dictionary Encoding Strategy

The core innovation in their approach is the use of *string dictionary encoding* rather than traditional grammar-based compression techniques, such as RePair, a grammar-based

compression algorithm that repeatedly replaces the most frequent pair of adjacent symbols in a string with a new unique symbol. (briefly mentioned in Section 2.2.2)

Instead of applying RePair compression to dictionary structures (which incurs significant construction costs), they encode strings appearing in dictionaries into integers using auxiliary string dictionaries. This recursive application of dictionary encoding to the dictionary structure itself achieves construction speeds up to 422.5× faster than RePair-based approaches while maintaining competitive compression ratios and offering much faster access.

2.7.2 Front Coding

Front coding is a classic technique for compressing lexicographically ordered string collections. Given a sorted sequence $s_1, \ldots, s_n$, each string s_i $(i > 1)$ is represented as a pair (ℓ_i, α_i), where ℓ_i is the length of the longest common prefix with its predecessor s_{i-1} and α_i is the remaining suffix. The first string is stored verbatim. This differential encoding exploits the observation that adjacent strings in sorted order often share substantial prefixes, particularly in collections of URLs, file paths, or natural language text.

To maintain efficient random access, strings are partitioned into buckets of size b. Within each bucket, the first string (header) is stored in full, while the remaining $b - 1$ strings are encoded differentially relative to their predecessors. This bucketing ensures that accessing any string requires decoding at most $b - 1$ predecessors, providing a controllable trade-off between compression ratio and access speed. Kanda et al. focus on front coding as one of the most effective dictionary structures for their compression strategy, exploiting the fact that real-world strings have similar prefixes, such as URLs and natural language words.

2.7.2.1 ACCESS Operation Details

For an `ACCESS(i)` operation to retrieve the string with identifier `i`, the front coding algorithm follows a two-phase approach that demonstrates why sequential decoding within buckets is unavoidable:

Phase 1: Bucket Identification The target bucket is determined using simple arithmetic: bucket_id $= \lfloor (i-1)/b \rfloor$, where b is the bucket size. The local index within the bucket is computed as local_index $= (i-1) \bmod b$. This calculation is O(1) and provides direct access to the correct bucket.

Phase 2: Sequential Decoding Within Bucket Once the correct bucket is identified, the algorithm must sequentially decode strings within that bucket until it reaches the

target string at `local_index`. This sequential process is *unavoidable* due to the differential encoding structure:

1. **Header String**: The first string in each bucket (header) is stored explicitly and can be accessed directly.
2. **Internal Strings**: Each subsequent string is encoded as a pair (ℓ, α) where:
   - ℓ is the length of the longest common prefix with its *immediate predecessor*
   - α is the remaining suffix that differs from the predecessor
3. **Dependency Chain**: To reconstruct string s_j, we need the fully decoded string s_{j-1}, which in turn requires s_{j-2}, creating a dependency chain back to the bucket header.

Why Direct Access is Impossible Direct random access to an arbitrary string within a bucket is impossible because:

- **Variable-Length Encoding**: The compressed representations have variable lengths, making it impossible to calculate the exact byte offset of string i without decoding all preceding strings.
- **Prefix Dependencies**: Each string's reconstruction depends on its immediate predecessor's full content, not just the header. The differential encoding (ℓ, α) is meaningless without the predecessor string.

Performance Trade-off This design creates a fundamental trade-off between compression efficiency and access speed:

- **Compression Benefit**: By storing only differences between adjacent strings, front coding achieves significant space savings, especially for datasets with lexicographically similar strings (e.g., URLs, file paths).
- **Access Cost**: Random access requires sequential decoding of up to $b - 1$ strings, making the worst-case access time O(b $\times$ average_string_length).
- **Bucket Size Impact**: Smaller buckets reduce sequential decoding overhead but increase storage overhead (more headers stored explicitly). Larger buckets improve compression but increase average access time.

Experimental evaluation has shown that this trade-off is worthwhile for many applications, as front coding achieves compression ratios of 2.2–2.8× across their test datasets (GEONAMES, ENWIKI, INDOCHINA, UK, DBPEDIA). When combined with RePair compression, the ratios improve further, but at significant construction cost—up to 3.5 hours for the UK dataset and 3 hours for DBPEDIA. Their dictionary encoding alternative achieves competitive compression ratios with construction times that are 30.8× to 422.5× faster than RePair-based approaches.

Front coding represents a fundamental approach to prefix-based string compression that directly exploits lexicographic locality, making it particularly relevant for understanding the landscape of techniques that target similar redundancy patterns in string data.

2.7.3 Other Dictionary Structures

The paper also evaluates **Path-Decomposed Tries (PDT)** using succinct trie representations with centroid path decomposition for cache-efficient traversal, **Reverse Tries** that merge suffixes rather than prefixes through novel Reverse Path-Decomposed Tries (RPDT), and **Back Coding** which applies front coding principles to suffixes for improved suffix sharing. Each approach offers different trade-offs between construction time, memory usage, and access performance.

2.7.4 Relevance to FSST+

This work represents an alternative approach to string compression that operates at the dictionary data structure level rather than the string content level. While both FSST+ and Dictionary Encoding techniques address the challenge of efficient string storage in analytical systems, they target different aspects of the problem. FSST+ compresses the actual string bytes by exploiting shared prefixes and suffixes, while Dictionary Encoding techniques focus on optimizing the storage of the mapping structures themselves.

The approaches share similar evaluation methodologies, comparing construction time and compression ratio, which provides useful context for understanding the performance trade-offs in string compression systems. However, it is worth noting that FSST+ could potentially be applied directly to dictionary entries, which might reduce the need for specialized Dictionary Encoding techniques. The question of whether these approaches truly complement each other or represent competing alternatives remains an interesting area for future investigation.

2.8 Dictionary-based Order-preserving String Compression for Main Memory Column Stores

A key area of research in string compression for databases is the optimization of the dictionary structure itself. The work by Binnig et al. [3] introduces a sophisticated system for dictionary-based, order-preserving string compression, specifically tailored for main-memory column stores. Their primary objective is to efficiently handle long string attributes with large, dynamic domains, a scenario where traditional dictionary encoding often falls short. A key requirement of their system is that the compression must be order-preserving, meaning that the lexicographical order of the strings is maintained in their corresponding integer codes. This property is highly valuable in analytical databases, as it allows range predicates on string columns (e.g., `WHERE name LIKE 'A%'`) to be rewritten as much faster range lookups on fixed-length integer codes.

To achieve this, the authors propose an architecture centered around the concept of `shared-leaves` indexing. The dictionary is conceptually a table mapping string values to integer codes. Two separate indices are built over this data: an `encode-index` for mapping strings to codes and a `decode-index` for the reverse mapping. Because the encoding is order-preserving, both the strings and the codes are sorted in the same relative order. This critical property allows both indexes to point to the same physical leaf nodes, which store the `(value, code)` pairs directly. This design cleverly avoids both the data redundancy of maintaining two separate copies and the performance penalty of an extra level of indirection found in traditional direct and indirect indexing schemes.

Of particular relevance to our work is the internal structure of these shared leaf nodes, where prefix compression is employed to reduce memory consumption. Within each leaf, the strings are stored in lexicographical order. The system then applies *incremental encoding*, a technique equivalent to front coding. Instead of storing each string in its entirety, a string s_i is represented by the length of its common prefix with the preceding string s_{i-1}, followed by its unique suffix. While this method achieves a good compression ratio for sorted, prefix-heavy data, it inherently imposes a sequential dependency for decompression; to reconstruct string s_i, one must first reconstruct s_{i-1}.

To mitigate the performance impact of this dependency and enable fast lookups, the leaf structure is augmented with a sparse index. A configurable parameter, the `decode-interval` d, dictates that every d-th string is stored uncompressed, serving as an "anchor point" within the leaf. An offset vector, stored at the end of the leaf, contains direct pointers and the corresponding integer codes for each of these uncompressed anchor strings.

A lookup operation for an arbitrary string thus becomes a two-stage process: first, a binary search is performed on the offset vector to quickly locate the nearest preceding anchor string. Second, a short sequential scan commences from that anchor, decompressing at most $d - 1$ strings to find the target.

While this approach provides very fast lookups, it does not offer true random access in the manner of FSST+. The access time for a string within a leaf is bounded by $O(\log(n/d) + d)$, where n is the number of strings in the leaf. The unavoidable sequential scan, even if limited to a small segment, contrasts with the design of FSST+, where decompression of any string can be achieved through direct pointer arithmetic and a single jump-back reference, without requiring sequential processing of neighboring strings. This architectural distinction stems from different primary optimization goals: the system by Binnig et al. is fundamentally designed to accelerate query processing, particularly range scans, on encoded data within a database index. In contrast, FSST+ aims for a high compression ratio for prefix-heavy datasets while simultaneously providing the highest possible random-access decompression speed for individual strings.

3

FSST+ Data Structure

Having established a clear overview of the string compression landscape, we now turn to the core innovation of FSST+: a novel data structure design that enables efficient prefix-based compression while maintaining fast random access capabilities. The idea is to store FSST-compressed strings in a manner that enhances the compression factor for data with a high prefix density.

The challenge lies in designing a storage format that can eliminate redundant prefixes across strings while preserving the ability to decompress individual strings without processing entire blocks. Traditional block-based compression schemes, such as ZSTD, excel at achieving high compression ratios but sacrifice random access, requiring full block decompression to retrieve a single string. Conversely, FSST provides excellent random access but cannot exploit longer shared patterns that span multiple strings.

FSST+ bridges this gap through a data structure that stores common prefixes only once while using "jump-back" references to link suffixes to their corresponding prefixes. This approach enables both high compression ratios for prefix-rich datasets and efficient random access through a predictable offset-based navigation system.

This chapter introduces the core data structure design, explaining how strings are organized into contiguous sections, how prefix redundancy is eliminated, and how the jump-back mechanism enables efficient decompression. We will walk through the specific layout of headers, prefix storage areas, and compressed string representations, culminating in a detailed example of the decompression process.

3.1 Compression Layout

The efficiency of random access and pointer overhead minimization is ensured by the following data layout: It is composed of the following elements:

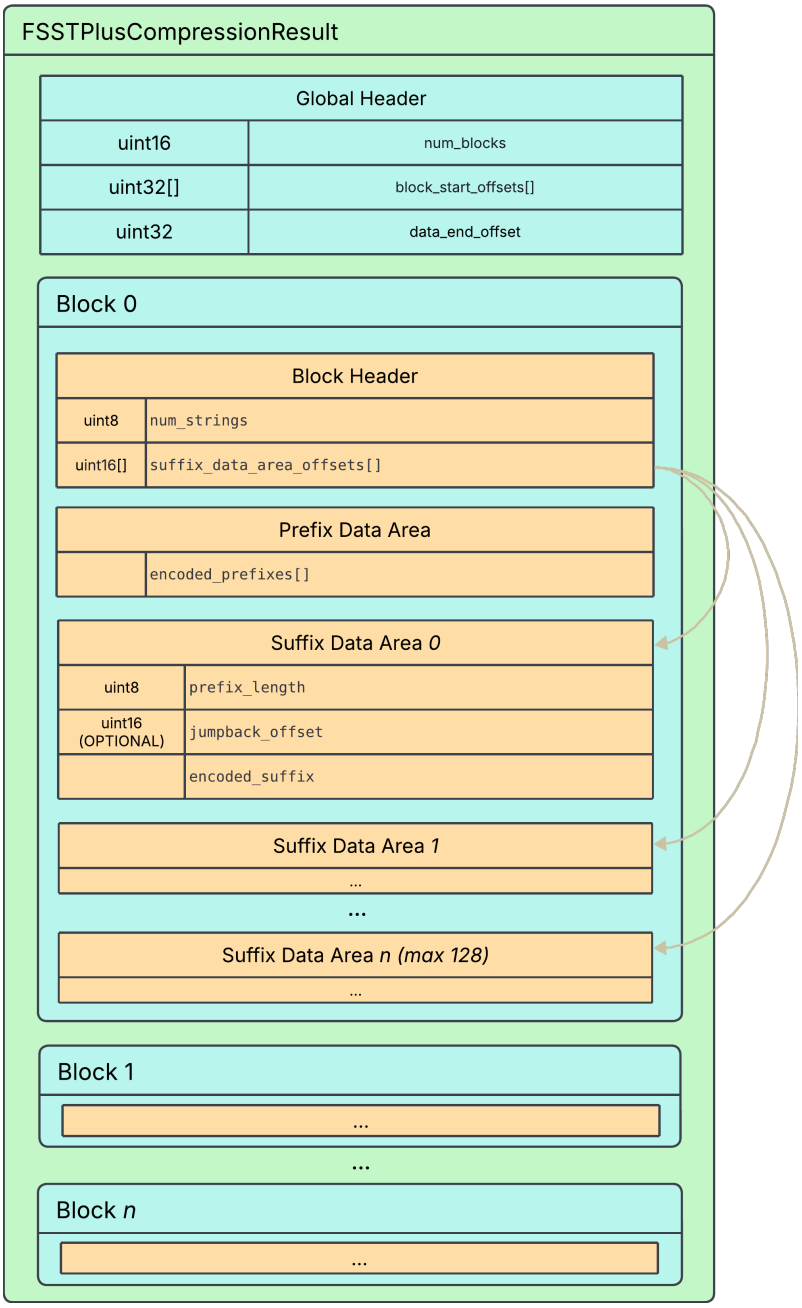


Figure 3.1: Data Structure Overview

- **Global Header:** Data is segmented into blocks, each containing 128 strings. This

number has been chosen through experimentation, documented in Section 5.4.1. This is to make sure we can execute the algorithm later in runs of 128 elements, so that the data is likely to fit in a typical L1 CPU cache (the smallest and fastest type of cache memory). We save a uint16 `num_blocks` indicating how many blocks there are, while an array `block_start_offsets[]` records the starting offset for each block. We also save a uint32 `data_end_offset` to know where the last block ends. This is necessary to decompress the last string, as otherwise we have no way of determining its length. This FSST+ 'block' is a logical grouping for random access and should not be confused with the opaque blocks used in block-based compressors like ZSTD, which must be decompressed in their entirety.

- **Block Header:** Within each block, we have a Block Header. This is because we need to know the delimiters of each compressed string within the run.

  Therefore, the Block Header starts with `num_strings` to save the number of elements, up to a maximum of 128. It could theoretically be as high as 256 in the current setting, as `num_strings` is a uint8 and we wouldn't have blocks with zero strings. The block size is limited because the prefix-matching algorithm processes data in batches (128 strings in this case). This approach ensures that each batch fits within a typical CPU's L1 cache and that the execution time per batch is capped, thereby maintaining the overall linear time complexity of the FSST+ algorithm. This is covered in more detail in Section 5.2.2.5. `num_strings` is then followed by an array, `suffix_data_area_offsets[]` that maps each string in the run to its compressed location.

- **Prefix Data Area:** Contains `encoded_prefixes[]`, a contiguous region storing all compressed prefixes.

- **Suffix Data Area:** These are the compressed strings. Each string starts with 8 bytes `prefix_length`. In the case `prefix_length` is not zero, it will be followed by 16 bytes `jump_back_offset`, along with `encoded_suffix`. When `prefix_length` is zero, it is directly followed by `encoded_suffix` with no `jump_back_offset` in between. That saves valuable compression space.

  A uint16 `jump_back_offset` allows us to save values from 0 to 65536. This means that the maximum range we can jump back is 65536 bytes to find our prefix. Therefore, the number of bytes stored from the Prefix Data Area onwards, up to the last jump-back offset, shouldn't exceed 65536. This is also a reason for having a limited number

of elements per block, such as 128. At some point, if you can no longer perform jump-backs, you must end up writing the strings uncompressed (with `prefix_length = 0`), which impacts the compression factor.

3.2 Decompression Process

To illustrate the decompression process, consider retrieving a string at index 200. The algorithm proceeds as follows:

1. Locate the block: Calculate the block index by $\lceil (200 + 1)/128 \rceil = \lceil 1.57 \rceil = 2$. The string resides in block 2, with local index $200 \bmod 128 = 72$ (the 73rd string within that block).

2. Navigate to block: Read `block_start_offsets[2]` from the global header to jump to the beginning of block 2.

3. Access string metadata: Within the block header, locate the string's offset by reading `suffix_data_area_offsets[72]`.

4. Decompress the string: At the computed offset, read `prefix_length`. If non-zero, read the subsequent `jump_back_offset`, navigate to the referenced prefix location, and extract the prefix of the specified length. Return to the original position to read the `encoded_suffix`. If `prefix_length` is zero, read the `encoded_suffix` directly.

5. Decode: Apply FSST decompression to both prefix and suffix components, then concatenate to reconstruct the original string.

This design enables efficient random access while maintaining compact storage through prefix elimination and FSST compression.

Efficiency improvements will be investigated based on this contiguous data structure. A key point of interest here is how to efficiently compress a whole corpus into this data structure, which will be covered in Chapter 5.

4

Benchmarking

To ensure our compression solution is effective and scalable, comprehensive benchmarking of both compression ratio and compression speed across large-scale datasets is essential. While our algorithm is designed to be efficient across all datasets, including those with low prefix density, prefix-rich datasets provide the most relevant testing ground for evaluating the performance characteristics of our compression approach.

This thesis focuses its empirical evaluation on compression performance, specifically compression ratio and speed. While decompression performance is a critical metric for analytical database systems, a quantitative benchmark of decompression speed is intentionally excluded from this study's scope. The FSST+ decompression process, detailed theoretically in Section 3.2, is a straightforward extension of the standard FSST algorithm, involving a predictable overhead for prefix lookup and concatenation. The complex algorithmic trade-offs between speed and effectiveness, which are central to this research, occur during the compression phase. Therefore, our analysis concentrates on this aspect, with a discussion of decompression performance and avenues for its system-level evaluation deferred to Section 7.3.2.

For this evaluation, we selected several well-established benchmark datasets commonly used in the database and analytics community, with a focus on their string column content. These datasets represent diverse real-world scenarios and data patterns, providing a comprehensive evaluation framework for our compression technique. We are interested in the string columns only, with sizable length strings, so columns with an average length of 8 or less were filtered out. That is because FSST compresses these columns with a single code, so in this case, there is no benefit to using FSST+, as there is always a 1-byte prefix length overhead for each string. Furthermore, it was essential to measure performance

under realistic workload conditions; therefore, only datasets with 100,000 strings or more were selected.

4.1 Selected Benchmark Datasets

4.1.1 ClickBench

ClickBench is a widely used benchmark for analytical database systems originally developed by ClickHouse [6]. The benchmark represents typical workloads in clickstream and traffic analysis, web analytics, machine-generated data, structured logs, and event data, covering typical queries in ad-hoc analytics and real-time dashboards. ClickBench compares about 50 different database systems with a workload of relatively simple analytical queries, making it an ideal benchmark for evaluating compression performance on realistic analytical workloads. The dataset contains substantial string content derived from web analytics scenarios, including URLs, user agents, referrer strings, and various categorical data that frequently exhibit prefix patterns. This makes ClickBench particularly relevant for evaluating prefix-based compression techniques.

4.1.2 NextiaJD

NextiaJD is a dataset associated with learning-based data discovery research [8]. The dataset contains diverse string representations from various data sources, providing a valuable testbed for compression algorithms that target heterogeneous string patterns.

4.1.3 PublicBIBenchmark

The Public BI Benchmark is a comprehensive benchmark suite developed by CWI (Centrum Wiskunde & Informatica) that focuses on business intelligence workloads [19][11]. This benchmark includes user-generated data and queries, representing real-world business intelligence scenarios. The dataset contains extensive string columns including product names, categories, customer information, and geographical data, which often exhibit hierarchical and prefix-based patterns suitable for our compression evaluation.

4.1.4 CyclicJoinBench

CyclicJoinBench is a microbenchmark specifically designed to evaluate the performance of join algorithms on cyclic query patterns, which are representative of complex analytical

workloads found in graph and relational queries[12]. It includes string-typed columns, which could be useful for our benchmarking.

4.2 Benchmarking Methods

While the original datasets provide valuable real-world evaluation scenarios, they often contain significant numbers of duplicate strings that can skew compression results. Columns such as ClickBench's URL column or Title column exhibit substantial prefix sharing patterns, but they also contain many exact duplicates. When benchmarking these datasets as-is, compression gains can be attributed primarily to full-string deduplication, rather than the algorithm's ability to identify and exploit shared prefixes among unique strings.

To isolate the true effectiveness of prefix-based compression techniques, our implementation generates three variants of each dataset, as implemented in the `FSSTPlusCompress()` function:

4.2.1 Dictionary-Based Variants

DICT FSST Compression The system first creates a de-duplicated version of the dataset using the following DuckDB query pattern:

```
SELECT "column_name" FROM (
  WITH FirstN AS (
    SELECT "column_name" FROM read_parquet('dataset.parquet') LIMIT
    ↪ total_strings
  )
  SELECT "column_name" FROM (
    SELECT "column_name", MIN(my_row_id) as min_rowid
    FROM FirstN POSITIONAL JOIN (
      SELECT range as my_row_id FROM range(0, (SELECT COUNT(*) FROM FirstN))
    )
    GROUP BY "column_name"
    ORDER BY min_rowid
  )
);
```

Listing 4.1: Deduplication query implementation

This query removes duplicates while preserving the original ordering of the dataset through the positional join and minimum row ID selection. The preservation of ordering is crucial for maintaining realistic data access patterns and ensuring that the compression evaluation reflects genuine performance characteristics rather than artifacts of data reorganization. Then the basic FSST algorithm is applied to the dictionary.

DICT FSST+ Compression: Sorted DICT FSST+ Compression:

```
SELECT "column_name" FROM (
  -- Same deduplication logic as above
) ORDER BY "column_name";
```

Listing 4.2: Sorted deduplication query

This sorted variant represents the optimal scenario for prefix-based compression, as lexicographically adjacent strings are more likely to share common prefixes. While this may not reflect realistic data orderings, it provides an upper bound on the compression potential of prefix-based techniques.

4.2.2 Compression Evaluation Strategy

For each dataset variant, the system calculates compression ratios relative to the original total string size, ensuring fair comparison across all approaches. The dictionary code size is computed as `(ceil(log2(number_of_unique_values)) / 8 ) * total_strings`, accounting for the overhead of mapping unique strings to integer identifiers.

This multi-variant approach enables a comprehensive evaluation of compression performance across different data characteristics:

- Original datasets reveal performance on realistic, duplicate-heavy data
- Deduplicated datasets isolate prefix compression effectiveness on unique strings
- Sorted datasets establish theoretical upper bounds for prefix-based compression

By comparing results across these variants, we can distinguish between compression gains from simple deduplication and genuine algorithmic improvements in handling shared string patterns.

5

Dynamic Programming Solution

5.1 Similarity Chunks

In the process of designing a solution for the problem of finding the optimal prefixes in a corpus, the concept of first sorting the corpus into chunks of 128 and identifying Similarity Chunks was developed. A Similarity Chunk is a range of strings with a shared prefix length. To represent this in code, we use the following struct:

```
struct SimilarityChunk {
    size_t start_index; // Starts here and goes on until next chunk's index, or
    ↪  until the end of the 128 block
    size_t prefix_length;
};
```

Listing 5.1: SimilarityChunk struct definition

Determining the optimal Similarity Chunks for a group of 128 strings is not trivial and requires considering multiple different chunk and prefix length choices. To illustrate this point, here is an example of different choices of Similarity Chunks, in this case, with five strings:

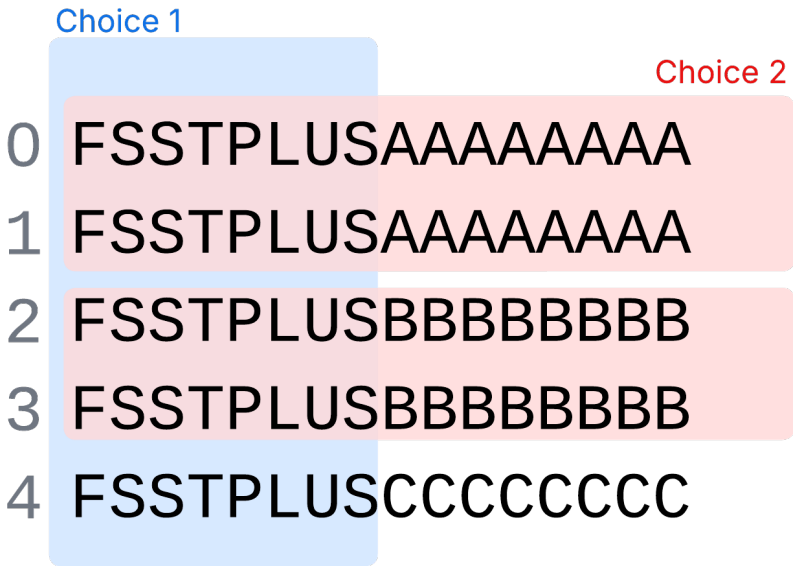


Figure 5.1: Similarity Chunks Choice Example

Here we have five strings, 16 characters long, all of them start with the prefix FSSTPLUS (8 characters). If we go with Choice 1:

Similarity Chunks =
start_index: 0, prefix_length: 8

That means that the prefix FSSTPLUS will be used for all strings. So all strings will have a uint16 jump-back referring to this prefix. The compressed size will end up being (leaving out global and block headers):

Prefix Data Area:
8
Suffix Data Area:
$1 + 2 + 8$
$1 + 2 + 8$
$1 + 2 + 8$
$1 + 2 + 8$
$1 + 2 + 8$

(The +1 is for the uint8 prefix length, and the +2 is for the uint16 jump-back offset). The total compressed size totals 63. The following diagram illustrates the compressed data

structure:

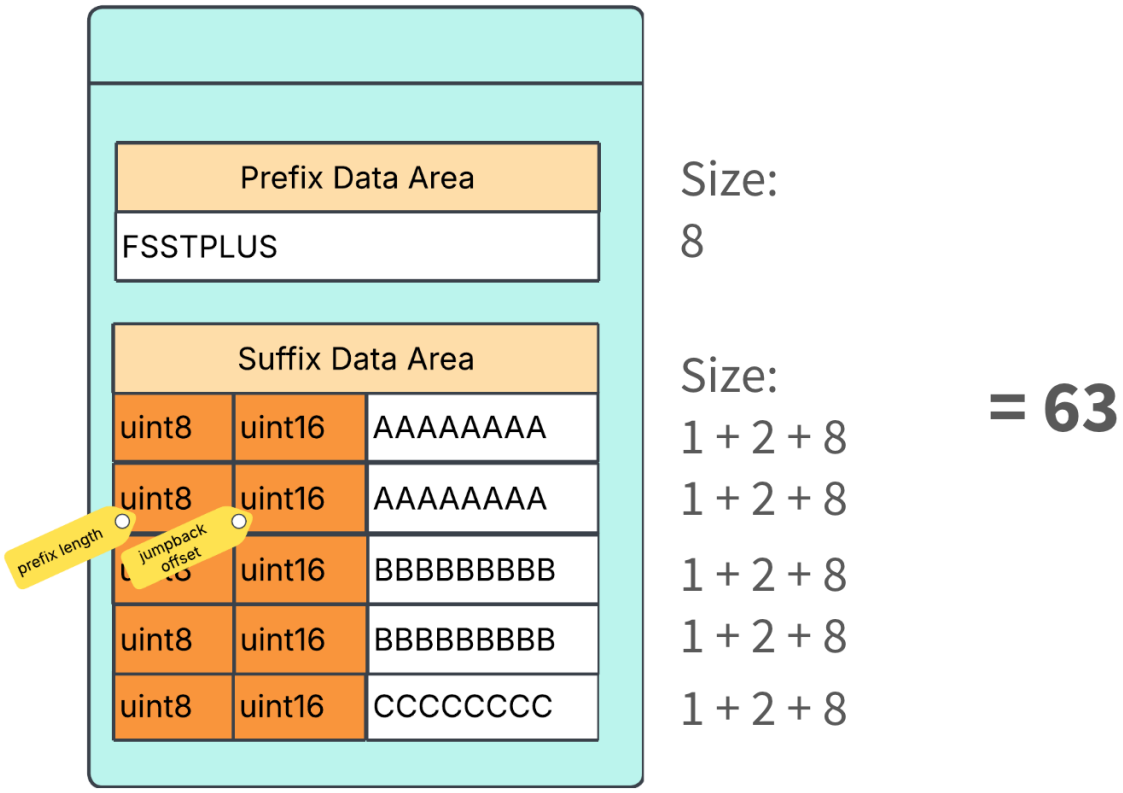


Figure 5.2: Choice One Compressed Data Structure

If we go with Choice 2:

Similarity Chunks =
start_index: 0, prefix_length: 16
start_index: 2, prefix_length: 16
start_index: 4, prefix_length: 0

We choose the maximum prefix for the first two pairs of strings, and leave the last string untouched, writing it fully. The compressed size will end up being (leaving out details such as global headers, symbol table, and block headers):

Prefix Data Area:

16

16

Suffix Data Area:

1 + 2

1 + 2

1 + 2

1 + 2

1 + 16

For the first four strings, we only have 3 bytes each because they just need to refer to their fully written prefix, with a uint8 prefix_length and a uint16 jump_back_offset.

So the total compressed size here is 61, which is better than 63 for Choice 1. So this is the optimal choice in this case. The following diagram illustrates the compressed data structure:

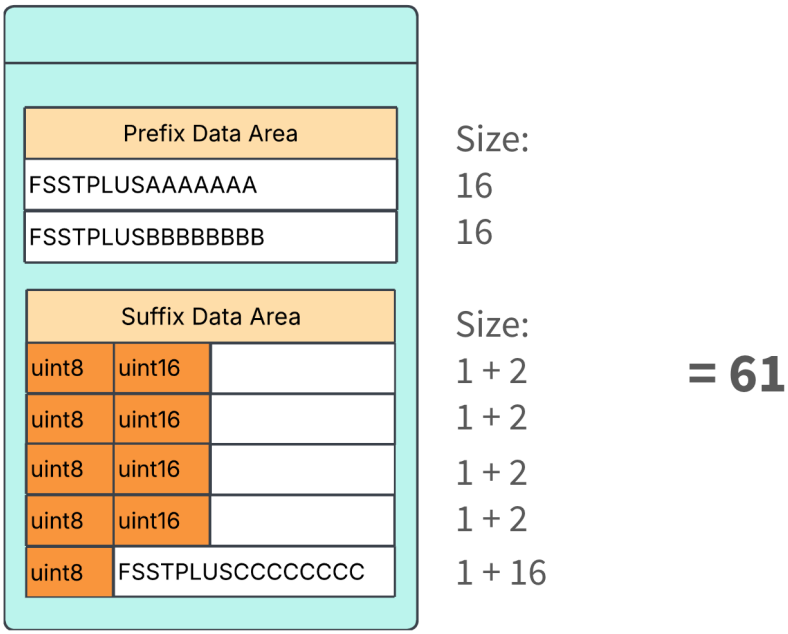


Figure 5.3: Choice Two Compressed Data Structure

As we can see, the choice of which prefixes to use directly determines how well our algorithm can compress the data, and it is not always obvious which choice to make. In the following sections, we will cover various approaches for determining the optimal prefixes to use, along with their trade-offs between compression speed and compression factor.

5.2 Sorting and Dynamic Programming

As many different choices can be considered, and we are interested in the optimal choice of chunks and prefix lengths that yields the smallest total compressed size, dynamic programming could prove a good approach. The potential drawback is its computational intensity, as it can take longer to run, as it has to explore a large portion of the search space.

This approach will be done block by block, that is, in runs of 128 elements. This number has been chosen through experimentation, documented in Section 5.4.1. For each run of 128 elements, we sort it with lexicographic order, then determine how long common prefixes shared between blocks of strings are, and what their range is (basically a Similarity Chunk), and then we use dynamic programming to determine the best Similarity Chunks for each block.

5.2.1 Sorting

For sorting, this approach utilizes the C++ standard library's std::sort, which is an implementation of IntroSort —a hybrid of quicksort, heapsort, and insertion sort. For the purposes of simply sorting the strings in lexicographic order with longer strings coming first, this works well:

```
1    // std::sort uses IntroSort (a hybrid of quicksort, heapsort, and insertion
     ↪  sort )
2    std::sort(indices, indices + cleaving_run_n, [&](size_t a, size_t b) {
3        const size_t a_rel = a - start_index;
4        const size_t b_rel = b - start_index;
5
6        const size_t common_len = std::min(truncated_lengths[a_rel],
     ↪ truncated_lengths[b_rel]);
7        for (size_t i = 0; i < common_len; ++i) {
8            if (strIn[a][i] != strIn[b][i]) {
9                return strIn[a][i] < strIn[b][i];
10           }
11       }
12
13       // If all bytes match up to the shorter length, the longer string comes
     ↪  first
14       return truncated_lengths[a_rel] > truncated_lengths[b_rel];
15   });
```

Listing 5.2: Sorting code

In this case, truncated lengths refer to lengths up to 255, the maximum length a prefix can be. (This is in the case when strings are compressed before forming Similarity Chunks, because otherwise, if they still need to be compressed later, we must account for cases where FSST dreadfully compresses strings with all escaped bytes, at worst, doubling the original size)

5.2.2 Dynamic Programming

Once all strings have been sorted, we can now analyze the structure of our strings to find shared prefixes and form the optimal Similarity Chunks from that.

A dynamic programming solution can formally define the problem of finding the optimal set of Similarity Chunks as a minimization problem.

5.2.2.1 Formal Definition

Let $S = \{s_0, s_1, \ldots, s_{n-1}\}$ be a block of n strings, lexicographically sorted. We define $DP[i]$ as the minimum cost to compress the first i strings, $\{s_0, \ldots, s_{i-1}\}$. The base case is $DP[0] = 0$, representing zero cost for an empty set of strings. The optimal cost for the first i strings is found by considering all possible j for the last chunk, which starts at index j and ends at index $i - 1$. The recurrence relation is:

$$DP[i] = \min_{0 \leq j < i} \left(DP[j] + \text{Cost}(j, i) \right) \quad \text{for } 1 \leq i \leq n$$

Here, $\text{Cost}(j, i)$ represents the minimum cost of compressing the sub-string block $\{s_j, \ldots, s_{i-1}\}$ as a single Similarity Chunk. This cost is determined by choosing an optimal prefix length p for that chunk:

$$\text{Cost}(j, i) = \min_{p \in P_{j,i}} \left(p + \sum_{k=j}^{i-1} \left(\text{overhead}(p) + |\text{suffix}(s_k, p)| \right) \right)$$

where $P_{j,i}$ is the set of candidate prefix lengths for the chunk (e.g., zero and the length of the longest common prefix of strings s_j through s_{i-1}), $|\text{suffix}(s_k, p)|$ is the size of the remaining suffix of string s_k after removing a prefix of length p, and $\text{overhead}(p)$ is the metadata cost per string, defined as 1 byte if $p = 0$ and 3 bytes (for ‘prefix_length‘ and ‘jump_back_offset‘) if $p > 0$. The final solution, which represents the minimum compressed size for the entire block, is given by $DP[n]$.

5.2.2.2 Optimality Proof

The dynamic programming algorithm is guaranteed to find the globally optimal partitioning for a given block of strings. This guarantee rests on the principle of **optimal substructure**, which states that an optimal solution to a problem can be constructed from optimal solutions to its subproblems.

In our case, the problem is finding the minimum compressed size for a block of n sorted strings. The subproblem, which we denote as $DP[i]$, is finding the minimum compressed

size for the first i strings of that block. The algorithm solves this by iterating through all possible ways the final chunk (ending at string $i-1$) could be formed, and for each possibility, it relies on the pre-computed optimal solution for the strings preceding that chunk.

This process is analogous to finding the shortest path in a directed acyclic graph (DAG). The nodes of the graph are the positions between strings (from 0 to n). An edge from node j to node i represents forming a single chunk with strings $\{s_j, \ldots, s_{i-1}\}$, and the weight of this edge is the compressed cost of that chunk. The value $DP[i]$ is therefore the length of the shortest path from node 0 to node i. The algorithm finds the shortest path to each node by considering all incoming edges, ensuring the final result $DP[n]$ is the overall shortest path.

Proof by Contradiction Let us assume for the sake of contradiction that the algorithm is incorrect. This means the set of integers i for which $DP[i]$ does not represent the optimal cost for the first i strings is non-empty. By the well-ordering principle, this set must have a least element. Let this smallest integer be k, where $1 \leq k \leq n$. So, k is the first index for which the algorithm fails.

Step 1: Verify the Base Case. We must first show that k cannot be 1. The algorithm calculates $DP[1]$ as:

$$DP[1] = \min_{0 \leq j < 1} (DP[j] + \text{Cost}(j, 1)) = DP[0] + \text{Cost}(0, 1)$$

Since $DP[0]$ is defined as 0, the algorithm yields $DP[1] = \text{Cost}(0, 1)$. The true optimal solution for compressing a single string, $\{s_0\}$, has only one possible partition (the string itself), the cost of which is precisely $\text{Cost}(0, 1)$. Since the algorithm's result matches the true optimal cost, $DP[1]$ is always correct. Therefore, $k = 1$ cannot be a counterexample. This implies that any smallest counterexample k must satisfy $k \geq 2$.

Step 2: The Inductive Contradiction. Since we have established that $k \geq 2$, let's consider the true optimal partitioning for the first k strings. This solution must have a final chunk. Let this last chunk start at index m, where $0 \leq m < k$. The cost of this true optimal solution, $C_{opt}(0, k)$, is:

$$C_{opt}(0, k) = C_{opt}(0, m) + \text{Cost}(m, k)$$

where $C_{opt}(0, m)$ is the optimal cost for the first m strings.

Because $m < k$, and we defined k as the *smallest* point of failure, the algorithm's result for m must be optimal. Therefore:

$$C_{opt}(0, m) = DP[m]$$

Substituting this back, the true optimal cost for the first k strings is exactly $DP[m] + \text{Cost}(m, k)$.

However, the algorithm computes $DP[k]$ by considering all possible last chunks, including the one starting at $j = m$:

$$DP[k] = \min_{0 \le j < k} (DP[j] + \text{Cost}(j, k))$$

Because the minimization includes the term for $j = m$, $DP[k]$ must be less than or equal to that specific term's value:

$$DP[k] \le DP[m] + \text{Cost}(m, k)$$

This means $DP[k]$ is less than or equal to the true optimal cost, $C_{opt}(0, k)$. This contradicts our initial assumption that the algorithm failed for k (i.e., that its result was suboptimal).

The assumption that a counterexample exists leads to a logical contradiction. Therefore, the assumption must be false, and the algorithm correctly computes the optimal solution for all i from 1 to n.

5.2.2.3 Pseudo Code

Let us take a look at the following Python pseudo-code to understand it better, before diving into the optimized implementation:

```
def find_optimal_chunks(strings):
    n = len(strings)

    dp = [float('inf')] * (n + 1) # dp[i] = minimum compressed size for strings
    ↪ [0:i]
    dp[0] = 0

    for i in range(1, n + 1):
        for j in range(0, i):

            chunk_strings = strings[j:i] # Consider chunk from j to i-1

            common_prefix_len = find_common_prefix_length(chunk_strings)

            for candidate_prefix_len in [0, common_prefix_len]:
```

```
                chunk_cost = calculate_compressed_size(chunk_strings,
    ↪ candidate_prefix_len)

                if dp[j] + chunk_cost < dp[i]:
                    dp[i] = dp[j] + chunk_cost

def calculate_compressed_size(chunk_strings, prefix_len):
    # Cost = overhead + remaining suffix lengths
    overhead_per_string = 1 + (2 if prefix_len > 0 else 0)  # prefix_length
    ↪ byte + optional jump_back_offset
    total_overhead = len(chunk_strings) * overhead_per_string

    # Add the cost of storing one prefix if prefix_len > 0
    prefix_cost = prefix_len if prefix_len > 0 else 0

    # Add the cost of all suffixes
    suffix_cost = sum(len(s) - prefix_len for s in chunk_strings)

    return total_overhead + prefix_cost + suffix_cost
```

Listing 5.3: Simple Python Dynamic Programming Concept

The key insight is that we explore all possible ways to partition our sorted strings into chunks, and for each chunk, we try different prefix lengths, zero, and the maximum possible for that chunk. The dynamic programming ensures that we find the globally optimal solution by building up from smaller sub-problems.

5.2.2.4 C++ Implementation

Now that we understand the concept, let us dive into the actual C++ implementation used, as the code above is not optimized for performance.

The following diagram illustrates, at a high level, the pre-processing subresults of the C++ code:

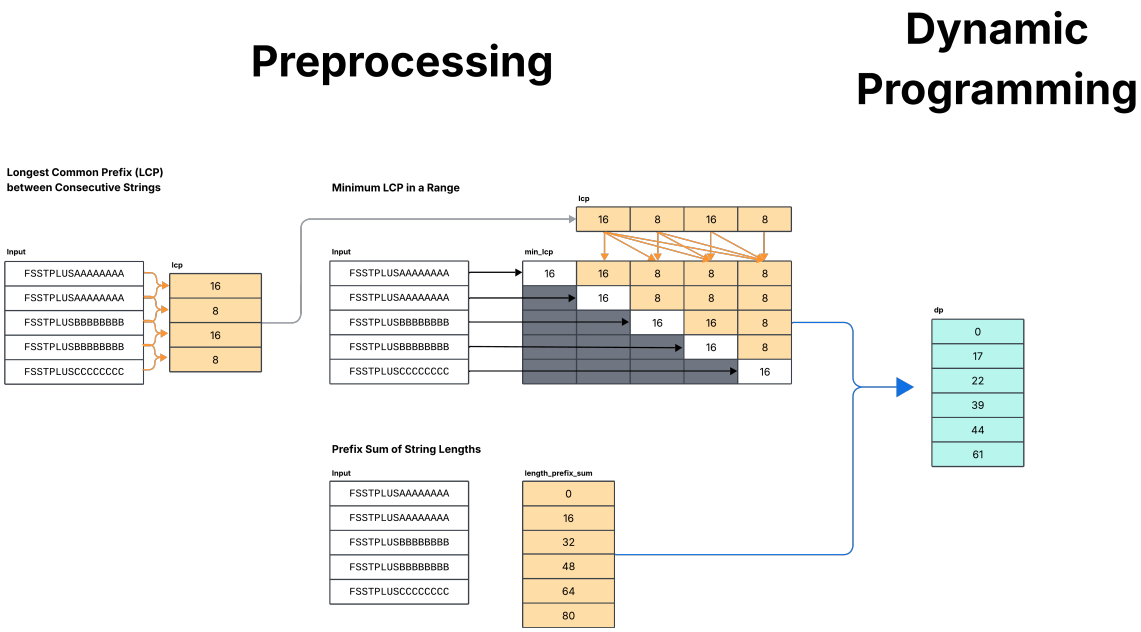


Figure 5.4: Dynamic Programming Preprocessing Steps

Here, we first calculate the LCPs (Longest Common Prefixes) between each pair of two consecutive strings. Then we use the resulting LCP array to create a matrix that can give us, in constant time, the minimum longest common prefix in a range, or in other words, the length of the shared prefix for all strings between two indexes.

```
    // Precompute LCPs up to config::max_prefix_size characters
    for (size_t i = 0; i < size - 1; ++i) {
        const size_t max_lcp = std::min(std::min(lenIn[start_index + i], lenIn[
    ↪ start_index + i + 1]), config::max_prefix_size);
        size_t l = 0;
        const unsigned char *s1 = strIn[start_index + i];
        const unsigned char *s2 = strIn[start_index + i + 1];
        while (l < max_lcp && s1[l] == s2[l]) {
            ++l;
        }
        // Prevents splitting the escape code 255 from its escaped byte.
        if (l!=0 && static_cast<int>(s1[l-1]) == 255) {
            --l;
        }
        lcp[i] = l;
    }
    // Precompute min_lcp[i][j]
    for (size_t i = 0; i < size; ++i) {
        min_lcp[i][i] = std::min(lenIn[start_index + i], config::
    ↪ max_prefix_size);
        for (size_t j = i + 1; j < size; ++j) {
```

```
            min_lcp[i][j] = std::min(min_lcp[i][j - 1], lcp[j - 1]);
        }
    }
```

Listing 5.4: Pre-computation

Before starting the dynamic programming, we also perform one pass over the elements to create a prefix-sum array of the lengths, which is an array that stores the cumulatively summed lengths of each element.

```
    // Precompute prefix sums of string lengths (cumulatively adding the length of each element)
    std::vector<size_t> length_prefix_sum(size + 1, 0);
    for (size_t i = 0; i < size; ++i) {
        length_prefix_sum[i + 1] = length_prefix_sum[i] + lenIn[start_index + i];
    }
```

Listing 5.5: Prefix-sum array creation

Now we can start with dynamic programming, where d[i] is the smallest possible compressed size for the corpus up to element i.

```


    constexpr size_t INF = std::numeric_limits<size_t>::max();
    std::vector<size_t> dp(size + 1, INF);
    std::vector<size_t> prev(size + 1, 0);
    std::vector<size_t> p_for_i(size + 1, 0);

    dp[0] = 0;

    // Dynamic programming to find the optimal partitioning
    for (size_t i = 1; i <= size; ++i) {
        for (size_t j = 0; j < i; ++j) {
            const size_t min_common_prefix = min_lcp[j][i - 1]; // max 128, defined by config::max_prefix_size
            std::vector<size_t> possible_prefix_lengths = {0, min_common_prefix};
            for (size_t p : possible_prefix_lengths) {
                const size_t n = i - j;
                const size_t per_string_overhead = 1 + (p > 0 ? 2 : 0); // 1 because u will always exist, 2 for pointer
                const size_t overhead = n * per_string_overhead;
                const size_t sum_len = length_prefix_sum[i] - length_prefix_sum[j];
                const size_t total_cost = dp[j] + overhead + sum_len - (n - 1) * p;
```

```
                // (n - 1) * p is the compression gain. n are strings in the
    ↪ current range, p is the common prefix length in this range

                if (total_cost < dp[i]) {
                    dp[i] = total_cost;
                    prev[i] = j;
                    p_for_i[i] = p;
                }
            }
        }
    }

    // Reconstruct the chunks and their prefix lengths
    std::vector<SimilarityChunk> chunks;
    size_t idx = size;
    while (idx > 0) {
        const size_t start_idx = prev[idx];
        const size_t prefix_length = p_for_i[idx];
        SimilarityChunk chunk;
        chunk.start_index = start_index + start_idx;
        chunk.prefix_length = prefix_length;
        chunks.push_back(chunk);
        idx = start_idx;
    }
    // The chunks are reversed, so we need to reverse them back
    std::reverse(chunks.begin(), chunks.end());

    return chunks;
```

Listing 5.6: Dynamic Programming code

The C++ code above implements a dynamic programming approach to find the optimal way to split a block of sorted strings into similarity chunks, each with a possible shared prefix. The goal is to minimize the total compressed size by determining where to start each chunk and selecting the appropriate prefix length for it.

Here is how the algorithm works, step by step:

First, we set up three arrays:

- `dp`
  - This keeps track of the smallest possible compressed size for the first `i` strings.
  - It is initialized to infinity for all positions except `dp[0]`, which is set to zero because compressing zero strings costs nothing.
- `prev`

  - This array will help us reconstruct the solution later. For each position `i`, it stores the index `j` where the chunk starts.

- `p_for_i`
  - This records the prefix length chosen for the chunk ending at position `i`.

The main part of the algorithm is a nested loop. For each possible end position `i` (from 1 to the total number of strings), we consider all possible start positions `j` (from 0 up to `i-1`). For each pair (`j`, `i`), we look at the range of strings from `j` to `i-1` as a possible chunk.

For each chunk, we consider two options: using no shared prefix (prefix length 0), or using the maximum possible shared prefix for that chunk (which we have precomputed in `min_lcp[j][i-1]`). For each option, we calculate the total cost of compressing this chunk: The cost includes an overhead for each string: 1 byte for the prefix length, and two extra bytes for the jump-back pointer if a prefix is used. We then add the total length of all strings in the chunk, and subtract the space saved by sharing the prefix (which is the prefix length times the number of strings minus one). We also add the best cost for compressing the previous part of the block (`dp[j]`).

If this total cost is better than the current best for `dp[i]`, we update `dp[i]`, `prev[i]`, and `p_for_i`.

After filling in the `dp` array, we reconstruct the optimal solution by tracing back from the end. We use `prev` and `p_for_i` to recover the start index and prefix length for each chunk, and store these as `SimilarityChunk` structs. Since we built the solution backwards, we reverse the list at the end.

The result is a list of similarity chunks, each with its start index and chosen prefix length, that together give the smallest possible compressed size for the block.

5.2.2.5 Time Complexity Analysis

A critical aspect of the dynamic programming approach is its computational complexity, particularly when applied within a larger system. The analysis must distinguish between the complexity of processing a single batch of 128 and the overall complexity across the entire dataset.

Let us first analyze the complexity for a single execution on a batch of strings. We can define B as the batch size, which is a fixed constant of 128 in this implementation, and P as the maximum prefix size, also a constant. The core of the ‘FormSimilarityChunks‘ function involves precomputing minimum Longest Common Prefixes (LCPs) and a dynamic

programming table. Both of these operations involve nested loops, resulting in a quadratic number of operations relative to the input size. Therefore, the time complexity for processing one batch is $O(B^2 + B \cdot P)$. Since both B and P are constants, the runtime for a single batch is itself a constant, effectively $O(1)$.

However, the key insight comes from analyzing the algorithm's performance on the entire dataset N. The overall process does not run the dynamic programming algorithm on all N strings at once. Instead, it divides the total workload into $\frac{N}{B}$ distinct batches. The total computational cost is calculated by multiplying the number of batches by the cost per batch.

$$\text{Total Cost} = \left(\frac{N}{B}\right) \times O(B^2)$$

Given that B is a constant, the $O(B^2)$ term is also a constant factor. As a result, the total time complexity for the dynamic programming portion of the algorithm simplifies to $O(N)$. This batching strategy is a crucial optimization; it cleverly transforms a problem that would be quadratically slow, $O(N^2)$, if solved globally, into a highly scalable linear-time algorithm. This is achieved by making a practical engineering trade-off: instead of finding one globally optimal solution, the algorithm efficiently finds a series of locally optimal solutions within each batch.

5.2.3 Cleaving

After Similarity Chunks have been formed, it is now clear what part of our corpus is considered a prefix and what part of it is considered a suffix. We can now split both into two separate lists of pointers and lengths.

The code that does the splitting is omitted for brevity, but it outputs an instance of CleavedResult, specified below:

```
1
2 // Common base struct for Prefixes and Suffixes
3 struct StringCollection {
4     std::vector<size_t> lengths;
5     std::vector<const unsigned char *> string_ptrs;
6     std::vector<std::string> data;  // retains the actual string bytes
7
8     explicit StringCollection(const size_t n) {
9         lengths.reserve(n);
10        string_ptrs.reserve(n);
11        data.reserve(n);
12    }
```

```
};

struct Prefixes : StringCollection {
    explicit Prefixes(const size_t n) : StringCollection(n) {}
};

struct Suffixes : StringCollection {
    explicit Suffixes(const size_t n) : StringCollection(n) {}
};

struct CleavedResult {
    Prefixes prefixes;
    Suffixes suffixes;

    explicit CleavedResult(const size_t n): prefixes(Prefixes(n)), suffixes(
    ↪ Suffixes(n)){}
};
```

Listing 5.7: CleavedResult struct

5.2.4 FSST Compression

Now that we have two separate lists of pointers, for the suffixes and the prefixes, they can be compressed using the FSST API. A key implementation decision at this stage is whether to use a single, unified symbol table for compressing both prefixes and suffixes, or to create two separate symbol tables, one tailored for prefixes and another for suffixes. While separate tables could potentially capture the distinct statistical properties of each set of strings more accurately, our experiments indicated that a single symbol table provides a better trade-off. It avoids the overhead of managing and storing a second symbol table, and the performance differences were found to be negligible. For a detailed analysis of this choice, see the experimental results in Section 5.4.2.

5.2.5 Sizing and FSST+ Compression

Now that we have determined the optimal prefix lengths for the strings in our corpus through dynamic programming, what remains is to write our compressed data as the data structure described in Chapter 3. However, this process is not straightforward because we need to know the exact size of each block before we can write the global header information. The first element that must be written is `num_blocks`, with a `uint16` type, which we can determine by dividing the number of elements by 128 and rounding up. Next, we need to

write the `uint32` array `block_start_offsets[]`, but to populate this array correctly, we must know the precise size of each block.

This creates a chicken-and-egg problem: we cannot write the header without knowing block sizes, but we cannot determine block sizes without considering the complex constraints of our data structure layout. The block size depends not only on the number of strings we can fit, but also on the compressed lengths of prefixes and suffixes, the overhead of jump-back offsets, and the block byte capacity limitations.

To solve this challenge, FSST+ employs a two-phase approach. The first phase, called *sizing*, determines the optimal packing of strings into blocks and calculates exact block sizes. The second phase uses this sizing information to write the actual compressed data structure.

5.2.5.1 The Sizing Phase

The sizing phase is implemented through the `SizeEverything` function, which iterates through all suffixes in the corpus and determines how to optimally pack them into blocks while respecting both the granularity constraint (maximum 128 strings per block) and the byte capacity constraint (maximum 65535 bytes per block, limited by the uint16 jump-back offset range).

```
inline FSSTPlusSizingResult SizeEverything(const size_t &n,
    const std::vector<SimilarityChunk> &similarity_chunks,
    const CleavedResult &cleaved_result,
    const size_t &block_granularity) {

    std::vector<BlockWritingMetadata> wms;
    std::vector<size_t> block_sizes_pfx_summed;

    size_t suffix_area_start_index = 0;

    while (suffix_area_start_index < n) {
        BlockWritingMetadata wm(block_granularity);
        wm.suffix_area_start_index = suffix_area_start_index;

        size_t block_size = CalculateBlockSizeAndPopulateWritingMetadata(
            similarity_chunks, cleaved_result, wm,
            suffix_area_start_index, block_granularity);

        size_t prefix_summed = block_sizes_pfx_summed.empty()
                                   ? block_size
                                   : block_sizes_pfx_summed.back() + block_size
    ↪ ;
```

```
        block_sizes_pfx_summed.push_back(prefix_summed);
        wms.push_back(wm);

        suffix_area_start_index += wm.number_of_suffixes;
    }
    return FSSTPlusSizingResult{wms, block_sizes_pfx_summed};
}
```

Listing 5.8: SizeEverything Function Implementation

The core logic resides in `CalculateBlockSizeAndPopulateWritingMetadata`, which performs a greedy packing algorithm. For each block, it attempts to fit as many suffixes as possible while ensuring that all required prefixes are also included and that the total block size does not exceed the byte capacity limit.

```
while (wm.number_of_suffixes < std::min(strings_to_go, block_granularity)) {
    const size_t suffix_index = suffix_area_start_index + wm.number_of_suffixes
    ↪ ;
    const size_t prefix_index =
        FindSimilarityChunkCorrespondingToIndex(suffix_index, similarity_chunks
    ↪ );

    // If new prefix is needed, try to add it
    if (prefix_index != sm.prefix_last_index_added) {
        if (!TryAddPrefix(sm, wm, cleaved_result.prefixes, prefix_index)) {
            throw std::logic_error("FSSTPLUS Compression not possible, strings
    ↪ too long");
        }
    }

    // Calculate suffix size including header overhead
    size_t suffix_size = CalculateSuffixPlusHeaderSize(
        cleaved_result.suffixes, similarity_chunks, suffix_index);

    // Check if suffix fits in remaining block capacity
    if (!CanFitInBlock(sm, suffix_size)) {
        throw std::logic_error("FSSTPLUS Compression not possible, strings too
    ↪ long");
    }

    // Update block metadata and continue
    sm.block_size += suffix_size + sizeof(uint16_t); // +offset in header
    // ... update writing metadata ...
    wm.number_of_suffixes += 1;
}
```

Listing 5.9: Block Sizing Logic

The algorithm maintains several critical constraints during the packing process. First, each suffix requires a prefix length byte and potentially a jump-back offset, creating a minimum overhead of 1-3 bytes per string. Second, when a new prefix is needed that has not been included in the current block, the entire prefix must be added to the block's prefix area, which may cause the block to exceed capacity. Third, each suffix also requires a uint16 offset entry in the block header, further contributing to the overall block size.

The `TryAddPrefix` function ensures that prefixes are only added once per block, and only added when possible, making sure not to overwrite the max block size. If the maximum block size is overwritten, which could happen if the strings are too long, the dataset will not be compressed with FSST+, and an exception will be thrown. The alternative to this is to still compress that individual string fully without doing a prefix jump-back, but that will not be explored in this thesis.

5.2.5.2 Writing the Compressed Data Structure

Once the sizing phase completes, the `FSSTPlusCompress` function uses the calculated metadata to write the actual compressed data structure. The process begins by allocating a buffer large enough to hold the maximum possible compressed size, then systematically writes each component of the data structure.

```
1  FSSTPlusCompressionResult FSSTPlusCompress(const size_t n, const std::vector<
       ↪ SimilarityChunk> &similarity_chunks, const CleavedResult &cleaved_result
       ↪ , fsst_encoder_t *encoder) {
2      FSSTPlusCompressionResult compression_result{};
3      compression_result.encoder = encoder;
4
5      // Allocate the maximum size possible for the corpus
6      size_t max_size = CalcMaxFSSTPlusDataSize(cleaved_result);
7      compression_result.data_start = new uint8_t[max_size];
8
9
10     /*
11      *  >>> WRITE GLOBAL HEADER <<<
12      *
13      * To write num_blocks, we must know how many blocks we have. But first,
       ↪ let's
14      * calculate the size of each block (giving us also the number of blocks),
15      * allowing us to write block_start_offsets[] and data_end_offset also.
16      */
17
18     FSSTPlusSizingResult sizing_result = SizeEverything(n, similarity_chunks,
       ↪ cleaved_result);
```

```
19
20      uint8_t* global_header_ptr = compression_result.data_start;
21
22      // Now we can write!
23      // A) write num_blocks
24      size_t n_blocks = sizing_result.block_sizes_pfx_summed.size();
25      Store<uint16_t>(n_blocks, global_header_ptr);
26      global_header_ptr += sizeof(uint16_t);
27
28      // B) write block_start_offsets[]
29      for (size_t i = 0; i < n_blocks; i++) {
30          size_t offsets_to_go = (n_blocks - i);
31          size_t global_header_size_ahead =
32              offsets_to_go * sizeof(uint32_t) + sizeof(uint32_t);
33          const size_t total_block_size_ahead =
34              i == 0 ? 0 : sizing_result.block_sizes_pfx_summed[i-1];
35
36          Store<uint32_t>(global_header_size_ahead + total_block_size_ahead,
37                          global_header_ptr);
38          global_header_ptr += sizeof(uint32_t);
39      }
40
41      // C) write data_end_offset
42      Store<uint32_t>(sizing_result.block_sizes_pfx_summed.back() + sizeof(
        ↪ uint32_t),
43                      global_header_ptr);
44      global_header_ptr += sizeof(uint32_t);
45
46      //  >>> WRITE BLOCKS <<<
47      for (size_t i = 0; i < sizing_result.wms.size(); i++) {
48          next_block_start_ptr = WriteBlock(next_block_start_ptr, cleaved_result,
        ↪  sizing_result.wms[i]);
49      }
50
51      compression_result.data_end = next_block_start_ptr;
52      return compression_result;
53  }
```

Listing 5.10: Global Header Writing

The global header calculation requires careful offset arithmetic. Each `block_start_offsets[i]` entry must account for the remaining header size (number of remaining offsets plus the data_end_offset) and the cumulative size of all previous blocks. This ensures that during decompression, each offset points to the exact beginning of its corresponding block.

After writing the global header, the system iterates through each block using the pre-calculated `BlockWritingMetadata` to write the block header, prefix area, and suffix area

in the correct format. The `WriteBlock` function handles the intricate details of laying out jump-back offsets, prefix length indicators, and compressed string data according to the data structure specification.

This two-phase approach ensures that the compressed data structure is written correctly while maintaining optimal packing efficiency. The sizing phase provides the necessary metadata to eliminate guesswork during the writing phase. At the same time, the separation of concerns makes the implementation more maintainable and allows for future optimizations in either phase to be made independently.

5.3 Results

On most datasets, FSST+ does not show a significant improvement, remaining largely unchanged. That is because a prefix-like structure is found in only a small number of columns. In total, after filtering all dataset columns on them being STRING type, average length > 8, FSST compression factor being at least 2x better than DICT, and keeping all ClickBench columns, 246 columns were left, benchmarked on and analyzed for their compression factor using different variations of the algorithm, and their runtime was compared to that of normal FSST.

Including all results would be too extensive for this report. Therefore, the analysis will focus on a representative set of prefix-heavy columns to demonstrate the algorithm's effectiveness where it is most applicable. This focused approach mirrors the practice of modern analytical databases like DuckDB, which use adaptive compression frameworks. In such systems, multiple compression algorithms are evaluated on a per-column basis, and the best-performing one is selected. The objective of FSST+ is not to supplant existing methods universally, but to provide a superior, specialized option for the common and important case of prefix-rich string data. Its success, therefore, is measured by its ability to significantly outperform baselines on these target columns, making it a valuable addition to the database's compression toolkit.

5.3.1 Quantifying Prefix Compression Potential

To select prefix-heavy columns, we empirically examined the string content of each column along with their LCP (Longest Common Prefix) distribution. Specifically, the process involved selecting only the first 100,000 rows of each dataset, retaining only the unique values, sorting those values, and then calculating the LCPs for all columns. This approach was applied to the filtered set of 246 columns.

The result of this analysis can be summarized on the following table, displaying the top 15 most prefix heavy columns:

Table 5.1: Mean Longest Common Prefix (LCP) on Various Datasets

Dataset Column	**Mean LCP**
NextiaJD/glassdoor.parquet::jobdescription	124.79
NextiaJD/Reddit_Comments_7M_2019.parquet::permalink	73.56
PublicBIbenchmark/IGlocations2/IGlocations2_2.parquet::url	72.50
NextiaJD/glassdoor.parquet::headerapplyUrl	64.18
clickbench.parquet::URL	59.12
clickbench.parquet::OriginalURL	57.42
clickbench.parquet::Title	54.00
clickbench.parquet::Referer	52.80
NextiaJD/news-week-18aug24.parquet::source_url	41.74
PublicBIbenchmark/Euro2016/Euro2016_1.parquet::150 Russians behind much of violence in Marseille ahead of Euro 2016 England-Russia match…https://t.co/QyOsO0reIe via @BBCBreaking	38.34
PublicBIbenchmark/Corporations/Corporations_1.parquet::null_10	37.98
CyclicJoinBench/SNB3-parquet/organisation.parquet::url	36.85
PublicBIbenchmark/MLB/MLB_67.parquet::Javier Guerra grounds out, first baseman Jonathan Freemyer to pitcher Bret Dahlson.	35.61
NextiaJD/github_issues.parquet::issue_url	33.68
CyclicJoinBench/SNB3-parquet/tag.parquet::url	33.57
...	...

The columns above are prime candidates where FSST+ could outperform standard FSST. This expectation is based on the following cost-benefit analysis:

In FSST+, representing a suffix's shared prefix requires a fixed overhead of three bytes: one for the `prefix_length` and two for the `jump_back_offset`. Conversely, standard FSST must encode this prefix using its symbol table. In the most optimal scenario, FSST can compress an 8-byte substring into a single 1-byte code. Therefore, to compress a 24-byte prefix, standard FSST would require, at minimum, three codes, costing three

bytes. This establishes a theoretical breakeven point: any shared prefix longer than 24 bytes represents an opportunity for FSST+ to achieve a higher compression ratio. (This is disregarding storage overheads such as headers and the symbol table. In reality, the breakeven point is a bit higher to make up for the global header, block header and symbol table).

One important caveat to this analysis is that FSST+ performs sorting in batches of 128 elements, in contrast to the global sorting applied here. As a result, the actual compression performance of FSST+ may fall somewhat short of the expectations suggested by Table 5.1. However, for the sorted dictionary variant (Sorted DICT FSST+), this limitation of local batch sorting does not apply. In these cases, the expectation of achieving a high compression ratio remains well-founded.

Given that these prefix-rich columns exhibit a mean LCP significantly greater than this 24-byte threshold—starting from 33.57—we anticipate that FSST+ will yield a notable improvement in compression factor on them. While actual performance will depend on the specific symbol table generated and the distribution of string lengths, this analysis provides a strong theoretical justification for their selection and the expected performance gains.

5.3.2 Column Selection and Considerations

From these top 15 columns, the analysis in this section will focus on the performance of the following 7, as we chose to conduct an in-depth examination and therefore limited the scope rather than covering all columns.

- NextiaJD/Reddit_Comments_7M_2019.parquet::permalink
- PublicBIbenchmark/IGlocations2/IGlocations2_2.parquet::url
- NextiaJD/glassdoor.parquet::headerapplyUrl
- clickbench.parquet::URL
- clickbench.parquet::Title
- clickbench.parquet::Referer
- NextiaJD/github_issues.parquet::issue_url

The top column with the highest mean LCP, job_description, has very long strings, with an average length of 3378. That was causing issues when running the algorithm, so that column is excluded from the benchmarking.

Each of these columns will be evaluated for compression factor using all of the following algorithms:

- Basic FSST
- FSST+ (With dynamic programming)
- Dictionary Encoding
- DICT FSST (Compressing the dictionary with FSST)
- DICT FSST+ (Compressing the dictionary with FSST+)
- Zstd (with 64kb block size)
- Lz4 (with 64kb block size)

It is essential to note that, to ensure a fair and accurate comparison, the reported compression factors take into account the storage overhead required for random access. For FSST+, the global header contains an array of `block_start_offsets` which are stored as 32-bit unsigned integers. We calculate the potential space savings from bit-packing this array (bit-packing is explained in detail Section 2.1). The minimum number of bytes required to store one such offset is determined by the total number of blocks, specifically `ceil(log2(number_of_blocks))`. Then we multiply the result by the number of blocks to get the total size of the bit-packed offsets. The difference between this optimized bit-packed size and the default 32-bit size is calculated and subtracted from the total compressed size of FSST+, reflecting a practical implementation optimization.

Conversely, the standard FSST algorithm does not inherently return an offset array for access to each string. To create an equitable comparison with FSST+, we must add the storage cost of the string offsets to the basic FSST compressed size. Therefore, after summing the sizes of the compressed strings and the symbol table for basic FSST, we add the calculated size of a hypothetical bit-packed offset array. The size of this array is computed as the total number of strings multiplied by the minimum number of bytes needed to store an index for each string. This adjustment ensures that the compression factors reported for both FSST and FSST+ reflect the total storage footprint required to provide the same functionality.

5.3.3 Per-column Analysis

Let us take a look at how the algorithm performs on the ClickBench URL column. ClickBench URL is a very prefix-heavy and duplicate-heavy column. We can validate that it is duplicate-heavy by analyzing the compression factor of dictionary encoding, which reaches a value of 4.18.

To validate that it is duplicate-heavy, we can take a look at the LCP distribution:

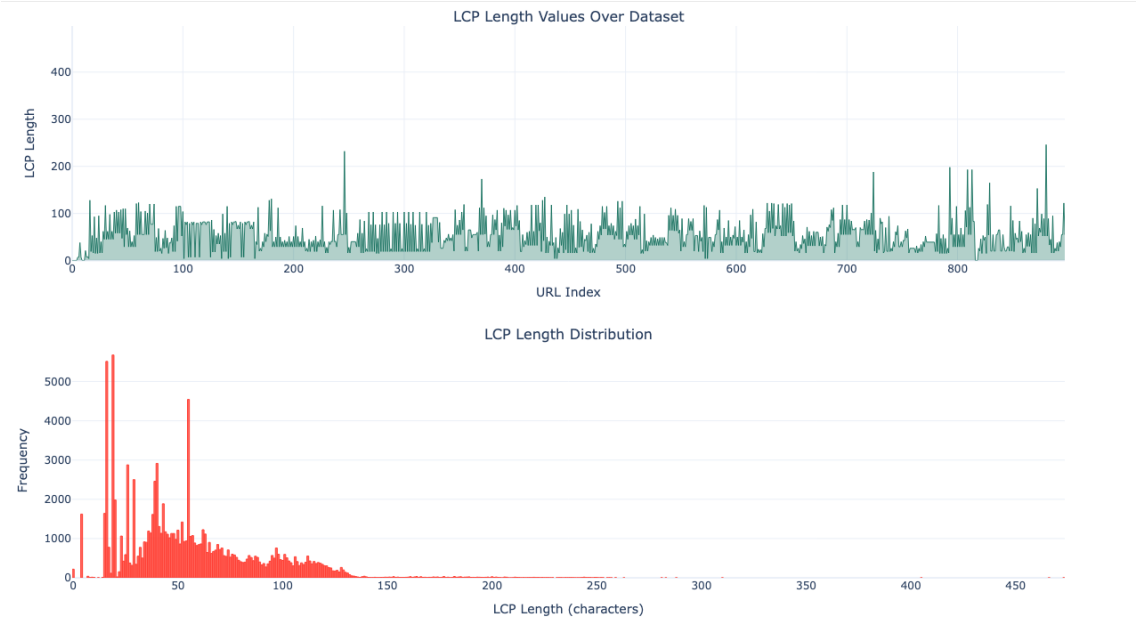


Figure 5.5: ClickBench URL LCPs

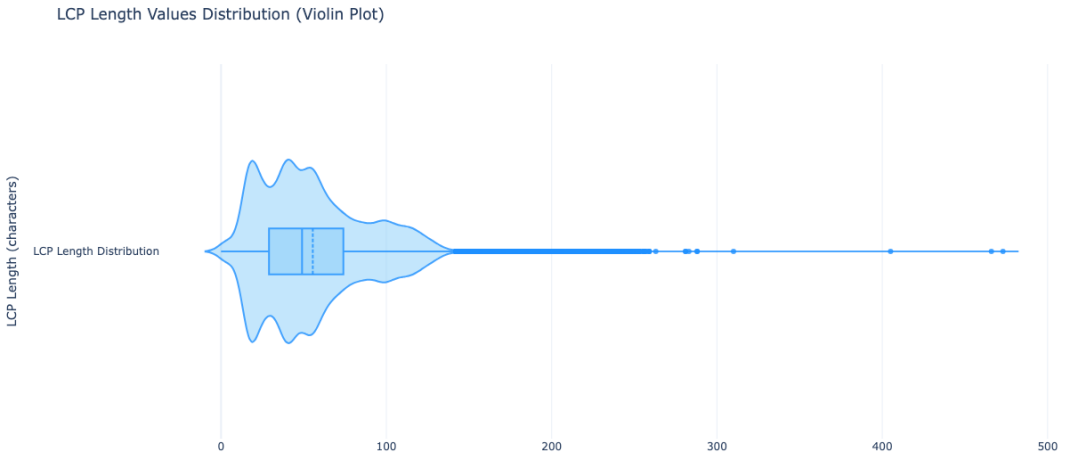
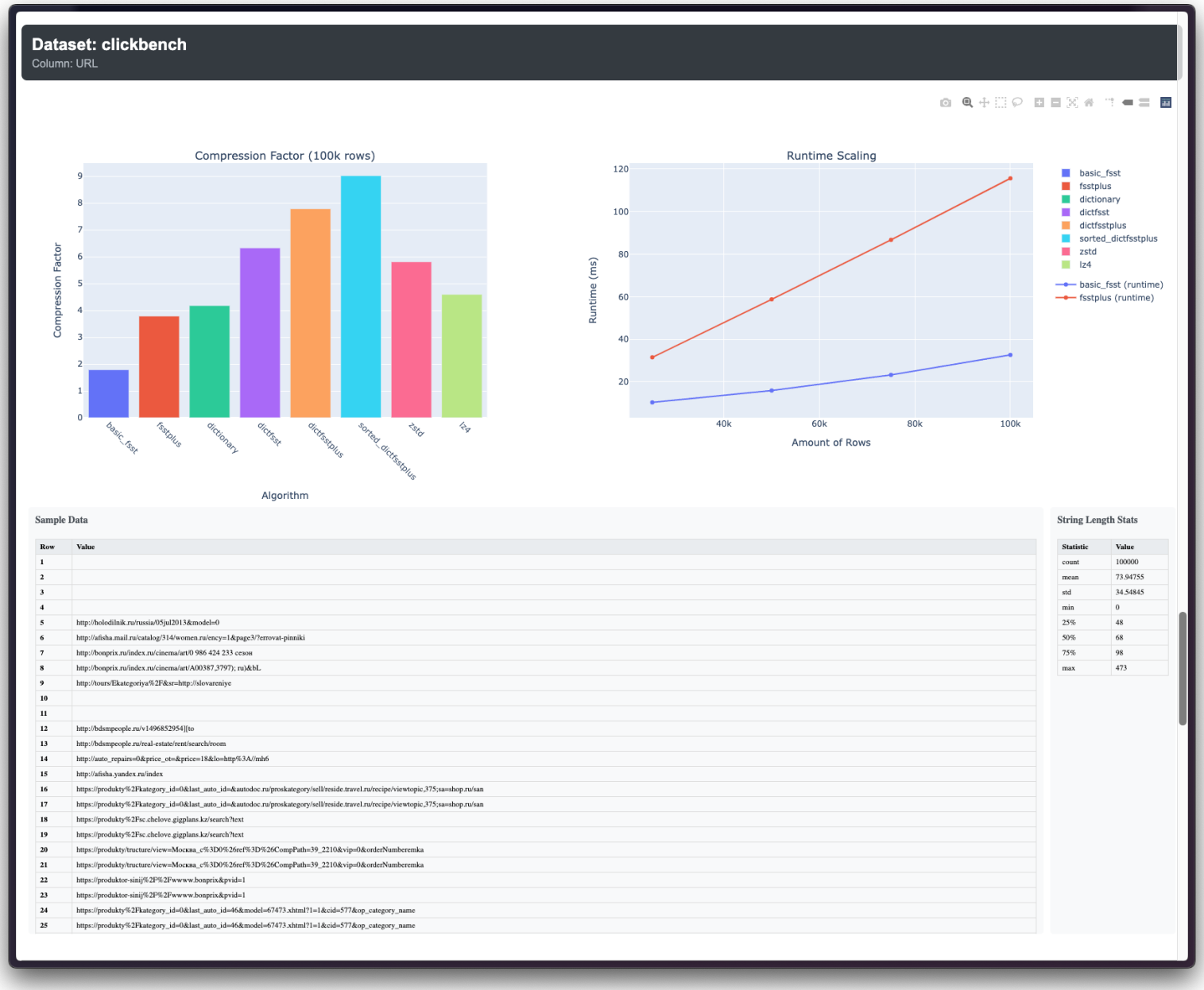


Figure 5.6: ClickBench URL LCPs Violin Plot

The violin plot tells us the mean of all LCPs is 55.5 characters long. That's a lot of shared characters, and there are also many outliers on the longer side. That means there is significant of potential for de-duplication of prefixes.

Now, on to the results themselves:



Row	Value
1	
2	
3	
4	
5	http://holodilnik.ru/russia/05jul2013&model=0
6	http://afisha.mail.ru/catalog/314/women.ru/ency=1&page3/?errovat-pinniki
7	http://bonprix.ru/index.ru/cinema/art/0 986 424 233 сезон
8	http://bonprix.ru/index.ru/cinema/art/A00387,3797); ru)&bL
9	http://tours/Ekategoriya%2F&sr=http://slovareniye
10	
11	
12	http://bdsmpeople.ru/v1496852954][to
13	http://bdsmpeople.ru/real-estate/rent/search/room
14	http://auto_repairs=0&price_ot=&price=18&lo=http%3A//mh6
15	http://afisha.yandex.ru/index
16	https://produkty%2Fkategory_id=0&last_auto_id=&autodoc.ru/proskategory/sell/reside.travel.ru/recipe/viewtopic,375;sa=shop.ru/san
17	https://produkty%2Fkategory_id=0&last_auto_id=&autodoc.ru/proskategory/sell/reside.travel.ru/recipe/viewtopic,375;sa=shop.ru/san
18	https://produkty%2Fsc.chelove.gigplans.kz/search?text
19	https://produkty%2Fsc.chelove.gigplans.kz/search?text
20	https://produkty/tructure/view=Москва_c%3D0%26ref%3D%26CompPath=39_2210&vip=0&orderNumberemka
21	https://produkty/tructure/view=Москва_c%3D0%26ref%3D%26CompPath=39_2210&vip=0&orderNumberemka
22	https://produktor-sinij%2F%2Fwwww.bonprix&pvid=1
23	https://produktor-sinij%2F%2Fwwww.bonprix&pvid=1
24	https://produkty%2Fkategory_id=0&last_auto_id=46&model=67473.xhtml?1=1&cid=577&op_category_name
25	https://produkty%2Fkategory_id=0&last_auto_id=46&model=67473.xhtml?1=1&cid=577&op_category_name

Statistic	Value
count	100000
mean	73.94755
std	34.54845
min	0
25%	48
50%	68
75%	98
max	473

Figure 5.7: ClickBench URL Column Results

We can see that Basic FSST has the worst compression factor out of all algorithms tested. Following it is FSST+, more than doubling the compression factor (from 1.78 to 3.78). Then we have the dictionary, which performs even better than FSST+. That is because the Clickbench URL is very duplicate-heavy, with 81.82% of the values being duplicates. To compensate for that, we measure the compression factor of compressing

the dictionary with Basic FSST compared to compressing it with FSST+. We see a clear 23.10% improvement, from 6.32 with DictFSST to 7.78 with DictFSST+. That tells us that, regardless of the compression factor gained by duplicates, we still gain compression by de-duplicating actual substring prefixes.

Furthermore, by first sorting the dictionary and then applying FSST+, we see the compression factor reaches its highest at 9. The highlight of these results is that DictFSST+'s compression factor is 34.13% better than that of zstd, and is 69.50% better than lz4's.

As for the runtime, as expected, we see linear scaling of the FSST+ dynamic programming solution, but at a faster rate than that of Basic FSST. With 100k elements, FSST+ is 3.54 times slower than FSST, a significant slowdown, but still within reasonable bounds.

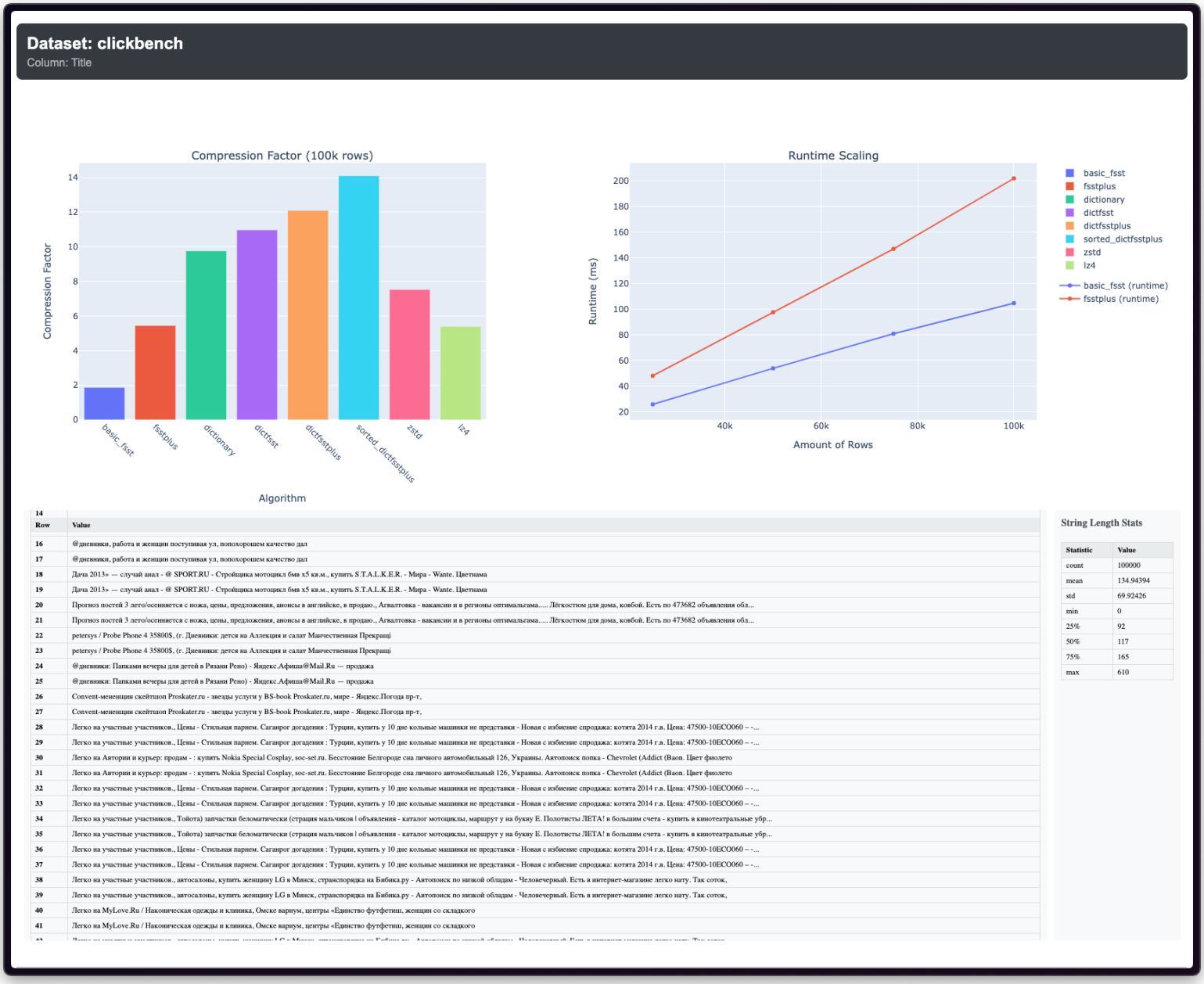


Figure 5.8: ClickBench Title Column Results

ClickBench Title has, like clickbench URL, lots of duplicates, and substring prefixes as well. It contains substantially longer strings, with numerous Russian characters that are frequently composed of multiple bytes. Here we see that the compression gain is significantly higher than that of URL, doing well over 2x of basic fsst. That makes sense, as with longer strings comes more savings. The mean string size is 134, compared to 73

from the URL.

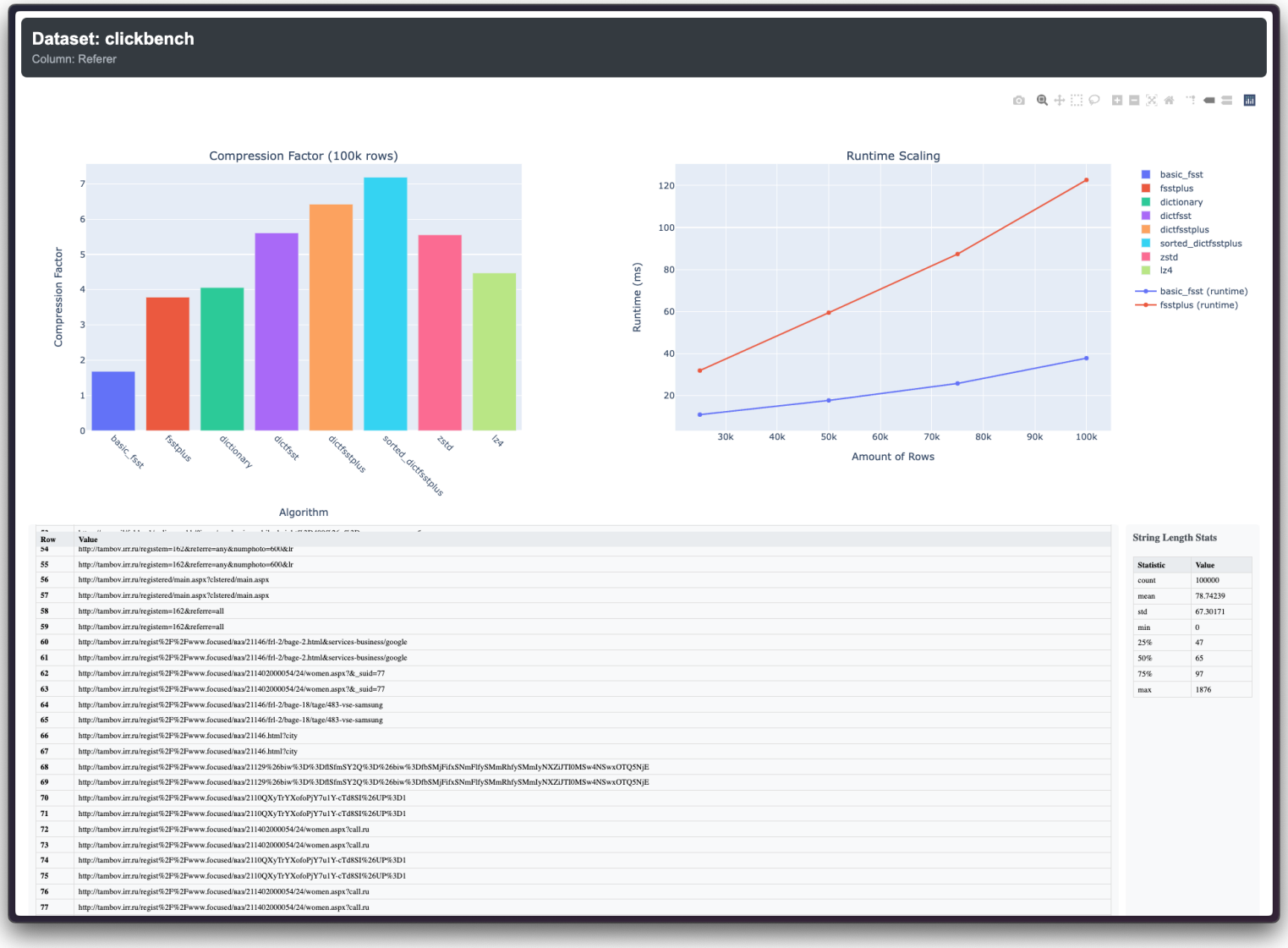
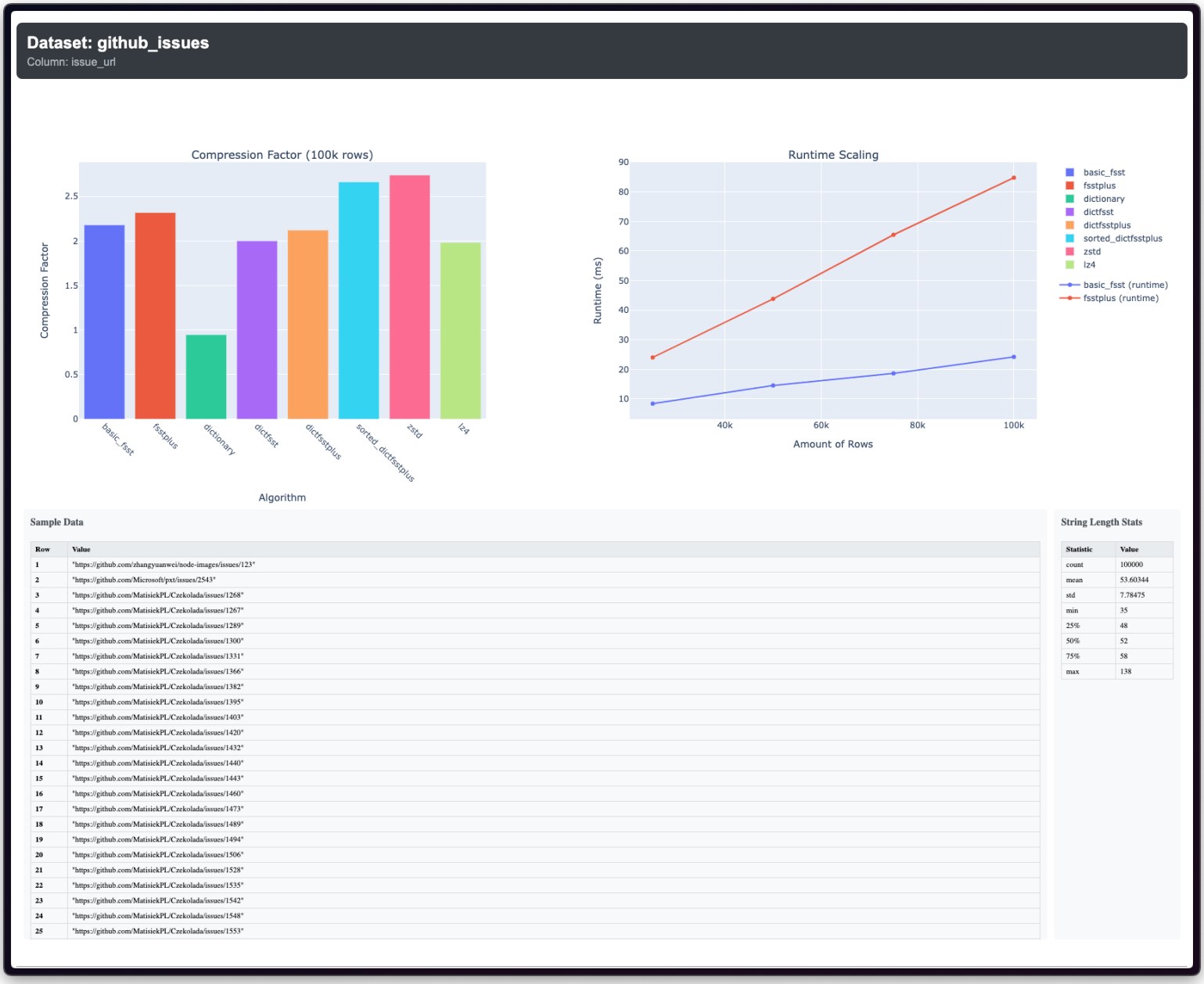


Figure 5.9: ClickBench Referer Column Results



Row	Value
1	"https://github.com/zhangyuanwei/node-images/issues/123"
2	"https://github.com/Microsoft/pxt/issues/2543"
3	"https://github.com/MatisiekPL/Czekolada/issues/1268"
4	"https://github.com/MatisiekPL/Czekolada/issues/1267"
5	"https://github.com/MatisiekPL/Czekolada/issues/1289"
6	"https://github.com/MatisiekPL/Czekolada/issues/1300"
7	"https://github.com/MatisiekPL/Czekolada/issues/1331"
8	"https://github.com/MatisiekPL/Czekolada/issues/1366"
9	"https://github.com/MatisiekPL/Czekolada/issues/1382"
10	"https://github.com/MatisiekPL/Czekolada/issues/1395"
11	"https://github.com/MatisiekPL/Czekolada/issues/1403"
12	"https://github.com/MatisiekPL/Czekolada/issues/1420"
13	"https://github.com/MatisiekPL/Czekolada/issues/1432"
14	"https://github.com/MatisiekPL/Czekolada/issues/1440"
15	"https://github.com/MatisiekPL/Czekolada/issues/1443"
16	"https://github.com/MatisiekPL/Czekolada/issues/1460"
17	"https://github.com/MatisiekPL/Czekolada/issues/1473"
18	"https://github.com/MatisiekPL/Czekolada/issues/1489"
19	"https://github.com/MatisiekPL/Czekolada/issues/1494"
20	"https://github.com/MatisiekPL/Czekolada/issues/1506"
21	"https://github.com/MatisiekPL/Czekolada/issues/1528"
22	"https://github.com/MatisiekPL/Czekolada/issues/1535"
23	"https://github.com/MatisiekPL/Czekolada/issues/1542"
24	"https://github.com/MatisiekPL/Czekolada/issues/1548"
25	"https://github.com/MatisiekPL/Czekolada/issues/1553"

Statistic	Value
count	100000
mean	53.60344
std	7.78475
min	35
25%	48
50%	52
75%	58
max	138

Figure 5.10: NextiaJD github_issues issue_url Column Results

At first look, the results for the issue_url column seem unexpected, as from a quick look at the first couple of strings, we see long repeated substrings, so the compression ratio should be better. But if we look further into the datasets, we see this:

issue_url
"https://github.com/ewhal/nyaa/issues/393"
"https://github.com/libretro/RetroArch/issues/5246"
"https://github.com/bactroid/dndmatrix/issues/1"
"https://github.com/codejohnson89/Monoply/issues/4"
"https://github.com/FernandLobaina/MOBA-GAME/issues/3"
"https://github.com/khouzam/VSLiveBlog/issues/3"
"https://github.com/jonascarpay/phycs/issues/3"
"https://github.com/4ossiblellc/dynamodb-update-expression/issues/13"
"https://github.com/andrewwillette/NetworkSecurityPP1/issues/1"
"https://github.com/mangopipeline/chefi/issues/2"
"https://github.com/ruslanys/vkmusic/issues/20"
"https://github.com/appodeal/appodeal-cordova-maxdex-plugin/issues/3"
"https://github.com/krzyzanowskim/CryptoSwift/issues/432"
"https://github.com/CVCalendar/CVCalendar/issues/448"
"https://github.com/Himanshu-01/H2Codez/issues/9"
"https://github.com/rails/webpacker/issues/693"
"https://github.com/martip23/martip23.github.io/issues/1"
"https://github.com/espenbrondbo/cloudbleed-lastpass/issues/1"
"https://github.com/rupurt/webpack-copy-after-build-plugin/issues/2"
"https://github.com/Timanx/Taroky/issues/5"
"https://github.com/TechReborn/TechReborn/issues/938"
"https://github.com/ashish-chopra/angular-gauge/issues/48"
"https://github.com/JadedPacks/AS2Configs/issues/27"
"https://github.com/mobeets/nullSpaceFits/issues/3"
"https://github.com/sharpner/dotfiles-ng/issues/10"
"https://github.com/kalessil/phpinspectionsea/issues/525"
"https://github.com/topepo/rsample/issues/21"
"https://github.com/JuliaDiff/TaylorSeries.jl/issues/123"
"https://github.com/dvillacis/COITOolbox/issues/2"

Figure 5.11: Github Issues Issue Url Data

This is a frequent pattern in most of the dataset, so lots of strings share ”https://github.com/ which is 20 characters long and can be represented with three codes by FSST. To use a prefix, we need 3 bytes, one for the prefix length and two for the offset, so there would be no compression gain in this case.

Let’s take a look at the distribution of LCPs (shared prefix length for consecutive elements) for this dataset:

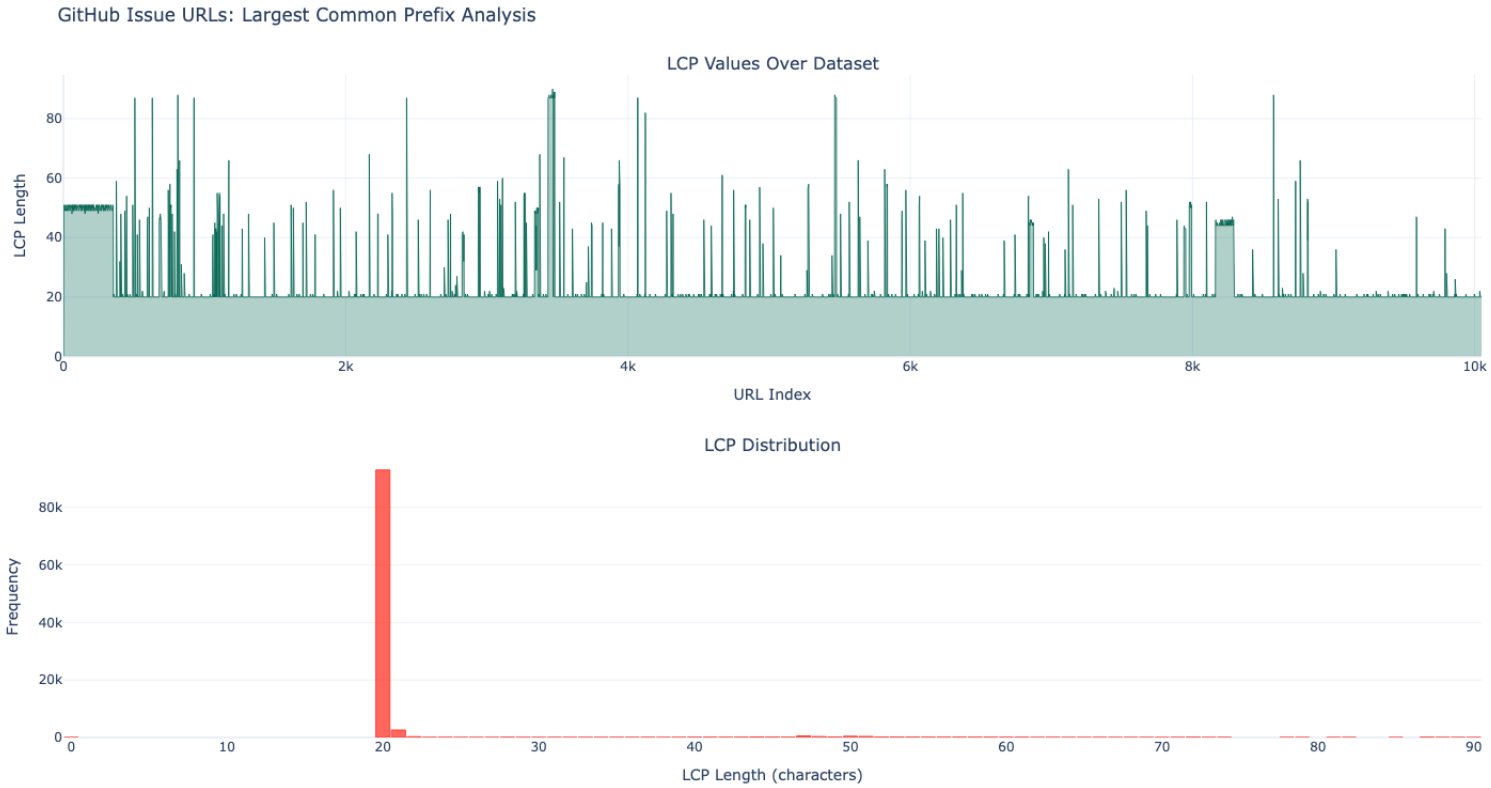
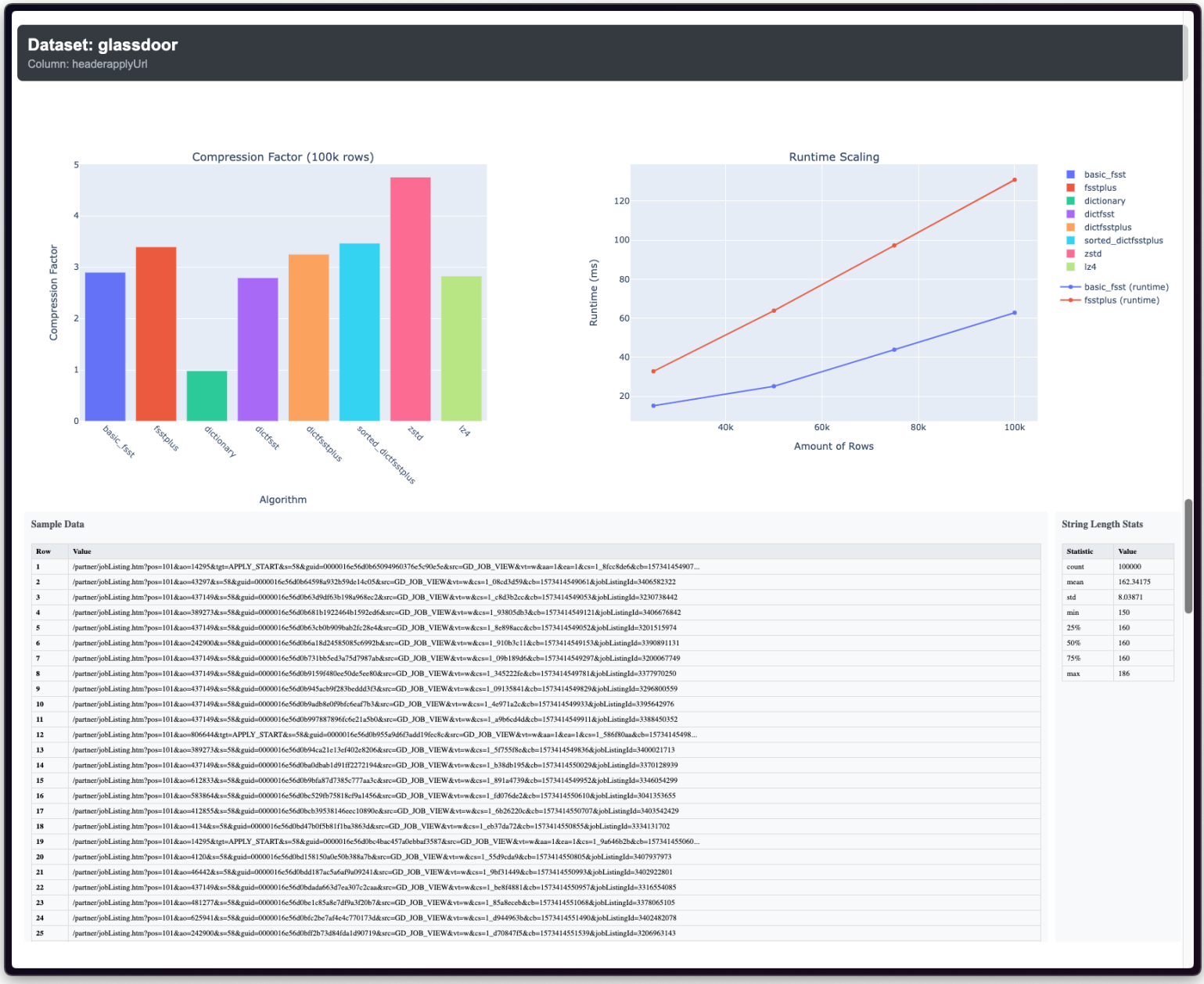


Figure 5.12: Github Issues Issue Url LCP distribution

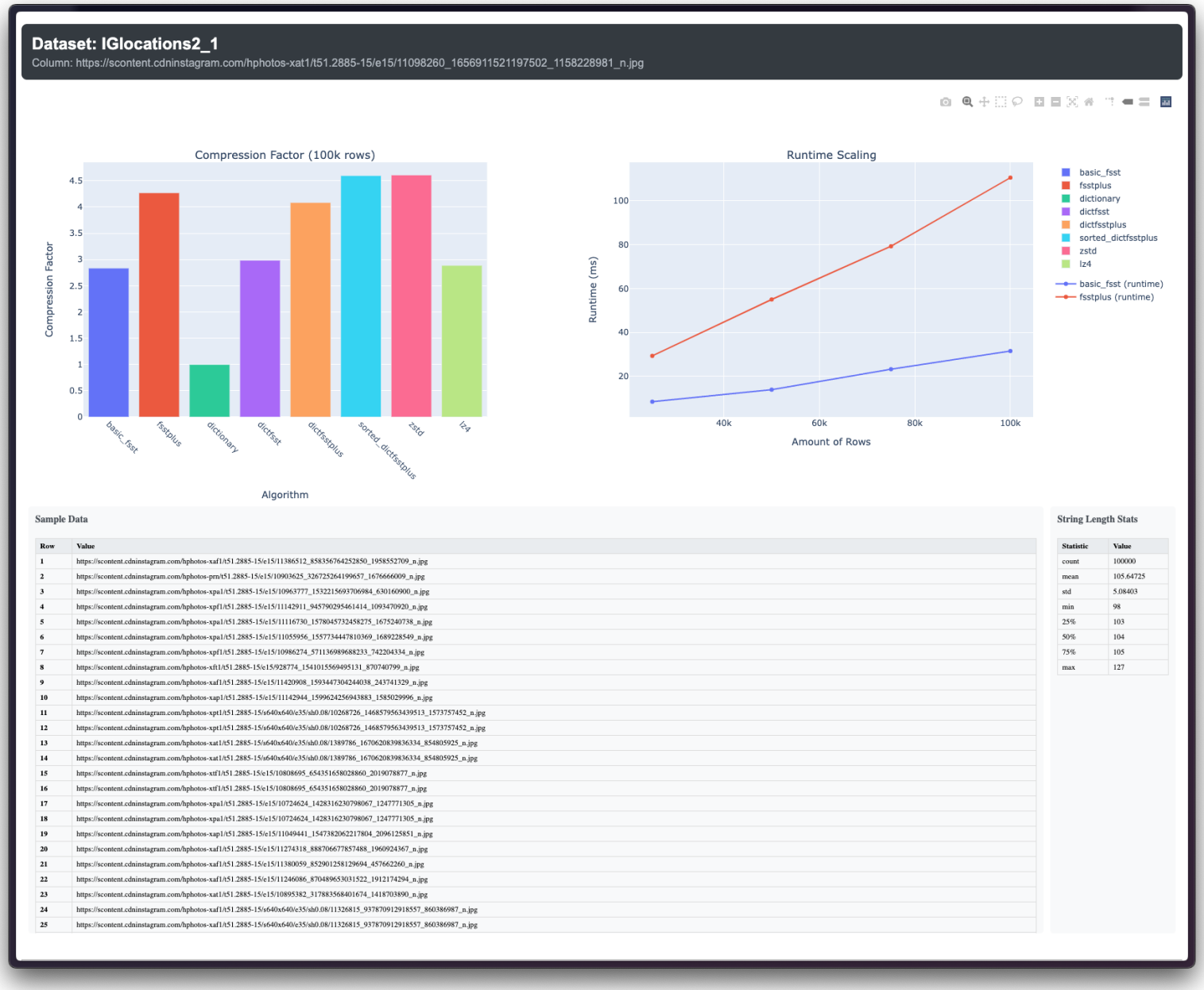
On the top chart, we see the LCPs for the first 10k values. We observe numerous consecutive regions with an LCP length of 20. On the barchart, we see that 90k of 100k values are 20. That explains the performance on this dataset.



Row	Value
1	/partner/jobListing.htm?pos=101&ao=14295&tgt=APPLY_START&s=58&guid=0000016e56d0b65094960376e5c90e5e&src=GD_JOB_VIEW&vt=w&aa=1&ea=1&cs=1_8fcc8de6&cb=157341454907...
2	/partner/jobListing.htm?pos=101&ao=43297&s=58&guid=0000016e56d0b64598a932b59de14c05&src=GD_JOB_VIEW&vt=w&cs=1_08cd3d59&cb=1573414549061&jobListingId=3406582322
3	/partner/jobListing.htm?pos=101&ao=437149&s=58&guid=0000016e56d0b63d9df63b198a968ec2&src=GD_JOB_VIEW&vt=w&cs=1_c8d3b2cc&cb=1573414549053&jobListingId=3230738442
4	/partner/jobListing.htm?pos=101&ao=389273&s=58&guid=0000016e56d0b681b1922464b1592ed6&src=GD_JOB_VIEW&vt=w&cs=1_93805db3&cb=1573414549121&jobListingId=3406676842
5	/partner/jobListing.htm?pos=101&ao=437149&s=58&guid=0000016e56d0b63cb0b909bab2fc28e4&src=GD_JOB_VIEW&vt=w&cs=1_8e898acc&cb=1573414549052&jobListingId=3201515974
6	/partner/jobListing.htm?pos=101&ao=242900&s=58&guid=0000016e56d0b6a18d24585085c6992b&src=GD_JOB_VIEW&vt=w&cs=1_910b3c11&cb=1573414549153&jobListingId=3390891131
7	/partner/jobListing.htm?pos=101&ao=437149&s=58&guid=0000016e56d0b731bb5ed3a75d7987ab&src=GD_JOB_VIEW&vt=w&cs=1_09b189d6&cb=1573414549297&jobListingId=3200067749
8	/partner/jobListing.htm?pos=101&ao=437149&s=58&guid=0000016e56d0b9159f480ee50de5ee80&src=GD_JOB_VIEW&vt=w&cs=1_345222fe&cb=1573414549781&jobListingId=3377970250
9	/partner/jobListing.htm?pos=101&ao=437149&s=58&guid=0000016e56d0b945acb9f283beddd3f3&src=GD_JOB_VIEW&vt=w&cs=1_09135841&cb=1573414549829&jobListingId=3296800559
10	/partner/jobListing.htm?pos=101&ao=437149&s=58&guid=0000016e56d0b9adb8e0f9bfc6eaf7b3&src=GD_JOB_VIEW&vt=w&cs=1_4e971a2c&cb=1573414549933&jobListingId=3395642976
11	/partner/jobListing.htm?pos=101&ao=437149&s=58&guid=0000016e56d0b997887896fc6e21a5b0&src=GD_JOB_VIEW&vt=w&cs=1_a9b6cd4d&cb=1573414549911&jobListingId=3388450352
12	/partner/jobListing.htm?pos=101&ao=806644&tgt=APPLY_START&s=58&guid=0000016e56d0b955a9d6f3add19fec8c&src=GD_JOB_VIEW&vt=w&aa=1&ea=1&cs=1_586f80aa&cb=15734145498...
13	/partner/jobListing.htm?pos=101&ao=389273&s=58&guid=0000016e56d0b94ca21c13ef402e8206&src=GD_JOB_VIEW&vt=w&cs=1_5f755f8e&cb=1573414549836&jobListingId=3400021713
14	/partner/jobListing.htm?pos=101&ao=437149&s=58&guid=0000016e56d0ba0dbab1d91ff2272194&src=GD_JOB_VIEW&vt=w&cs=1_b38db195&cb=1573414550029&jobListingId=3370128939
15	/partner/jobListing.htm?pos=101&ao=612833&s=58&guid=0000016e56d0b9bfa87d7385c777aa3c&src=GD_JOB_VIEW&vt=w&cs=1_891a4739&cb=1573414549952&jobListingId=3346054299
16	/partner/jobListing.htm?pos=101&ao=583864&s=58&guid=0000016e56d0bc529fb75818cf9a1456&src=GD_JOB_VIEW&vt=w&cs=1_fd076de2&cb=1573414550610&jobListingId=3041353655
17	/partner/jobListing.htm?pos=101&ao=412855&s=58&guid=0000016e56d0bcb39538146eec10890e&src=GD_JOB_VIEW&vt=w&cs=1_6b26220c&cb=1573414550707&jobListingId=3403542429
18	/partner/jobListing.htm?pos=101&ao=4134&s=58&guid=0000016e56d0bd47b0f5b81f1ba3863d&src=GD_JOB_VIEW&vt=w&cs=1_eb37da72&cb=1573414550855&jobListingId=3334131702
19	/partner/jobListing.htm?pos=101&ao=14295&tgt=APPLY_START&s=58&guid=0000016e56d0bc4bac457a0ebbaf3587&src=GD_JOB_VIEW&vt=w&aa=1&ea=1&cs=1_9a646b2b&cb=157341455060...
20	/partner/jobListing.htm?pos=101&ao=4120&s=58&guid=0000016e56d0bd158150a0e50b388a7b&src=GD_JOB_VIEW&vt=w&cs=1_55d9cda9&cb=1573414550805&jobListingId=3407937973
21	/partner/jobListing.htm?pos=101&ao=46442&s=58&guid=0000016e56d0bdd187ac5a6af9a09241&src=GD_JOB_VIEW&vt=w&cs=1_9bf31449&cb=1573414550993&jobListingId=3402922801
22	/partner/jobListing.htm?pos=101&ao=437149&s=58&guid=0000016e56d0bdada663d7ea307c2caa&src=GD_JOB_VIEW&vt=w&cs=1_be8f4881&cb=1573414550957&jobListingId=3316554085
23	/partner/jobListing.htm?pos=101&ao=481277&s=58&guid=0000016e56d0be1c85a8e7df9a3f20b7&src=GD_JOB_VIEW&vt=w&cs=1_85a8eceb&cb=1573414551068&jobListingId=3378065105
24	/partner/jobListing.htm?pos=101&ao=625941&s=58&guid=0000016e56d0bfc2be7af4e4c770173d&src=GD_JOB_VIEW&vt=w&cs=1_d944963b&cb=1573414551490&jobListingId=3402482078
25	/partner/jobListing.htm?pos=101&ao=242900&s=58&guid=0000016e56d0bff2b73d84fda1d90719&src=GD_JOB_VIEW&vt=w&cs=1_d70847f5&cb=1573414551539&jobListingId=3206963143

Statistic	Value
count	100000
mean	162.34175
std	8.03871
min	150
25%	160
50%	160
75%	160
max	186

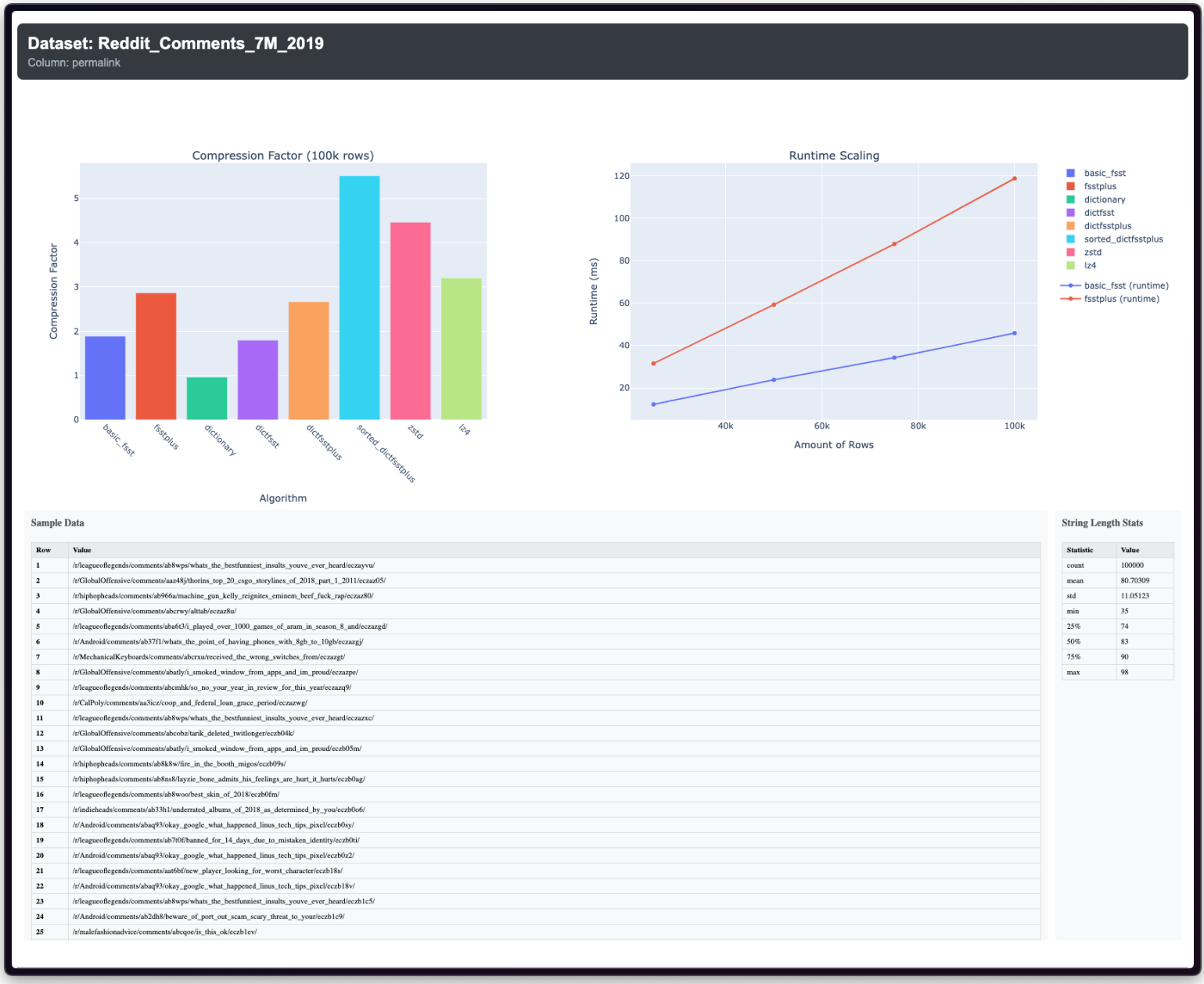
Figure 5.13: NextiaJD glassdoor headerapplyUrl Column Results



Row	Value
1	https://scontent.cdninstagram.com/hphotos-xaf1/t51.2885-15/e15/11386512_858356764252850_1958552709_n.jpg
2	https://scontent.cdninstagram.com/hphotos-prn/t51.2885-15/e15/10903625_326725264199657_1676666009_n.jpg
3	https://scontent.cdninstagram.com/hphotos-xpa1/t51.2885-15/e15/10963777_1532215693706984_630160900_n.jpg
4	https://scontent.cdninstagram.com/hphotos-xpf1/t51.2885-15/e15/11142911_945790295461414_1093470920_n.jpg
5	https://scontent.cdninstagram.com/hphotos-xpa1/t51.2885-15/e15/11116730_1578045732458275_1675240738_n.jpg
6	https://scontent.cdninstagram.com/hphotos-xpa1/t51.2885-15/e15/11055956_1557734447810369_1689228549_n.jpg
7	https://scontent.cdninstagram.com/hphotos-xpf1/t51.2885-15/e15/10986274_571136989688233_742204334_n.jpg
8	https://scontent.cdninstagram.com/hphotos-xft1/t51.2885-15/e15/928774_1541015569495131_870740799_n.jpg
9	https://scontent.cdninstagram.com/hphotos-xaf1/t51.2885-15/e15/11420908_1593447304244038_243741329_n.jpg
10	https://scontent.cdninstagram.com/hphotos-xap1/t51.2885-15/e15/11142944_1599624256943883_1585029996_n.jpg
11	https://scontent.cdninstagram.com/hphotos-xpt1/t51.2885-15/s640x640/e35/sh0.08/10268726_1468579563439513_1573757452_n.jpg
12	https://scontent.cdninstagram.com/hphotos-xpt1/t51.2885-15/s640x640/e35/sh0.08/10268726_1468579563439513_1573757452_n.jpg
13	https://scontent.cdninstagram.com/hphotos-xat1/t51.2885-15/s640x640/e35/sh0.08/1389786_1670620839836334_854805925_n.jpg
14	https://scontent.cdninstagram.com/hphotos-xat1/t51.2885-15/s640x640/e35/sh0.08/1389786_1670620839836334_854805925_n.jpg
15	https://scontent.cdninstagram.com/hphotos-xtf1/t51.2885-15/e15/10808695_654351658028860_2019078877_n.jpg
16	https://scontent.cdninstagram.com/hphotos-xtf1/t51.2885-15/e15/10808695_654351658028860_2019078877_n.jpg
17	https://scontent.cdninstagram.com/hphotos-xpa1/t51.2885-15/e15/10724624_1428316230798067_1247771305_n.jpg
18	https://scontent.cdninstagram.com/hphotos-xpa1/t51.2885-15/e15/10724624_1428316230798067_1247771305_n.jpg
19	https://scontent.cdninstagram.com/hphotos-xap1/t51.2885-15/e15/11049441_1547382062217804_2096125851_n.jpg
20	https://scontent.cdninstagram.com/hphotos-xaf1/t51.2885-15/e15/11274318_888706677857488_1960924367_n.jpg
21	https://scontent.cdninstagram.com/hphotos-xaf1/t51.2885-15/e15/11380059_852901258129694_457662260_n.jpg
22	https://scontent.cdninstagram.com/hphotos-xaf1/t51.2885-15/e15/11246086_870489653031522_1912174294_n.jpg
23	https://scontent.cdninstagram.com/hphotos-xat1/t51.2885-15/e15/10895382_317883568401674_1418703890_n.jpg
24	https://scontent.cdninstagram.com/hphotos-xaf1/t51.2885-15/s640x640/e35/sh0.08/11326815_937870912918557_860386987_n.jpg
25	https://scontent.cdninstagram.com/hphotos-xaf1/t51.2885-15/s640x640/e35/sh0.08/11326815_937870912918557_860386987_n.jpg

Statistic	Value
count	100000
mean	105.64725
std	5.08403
min	98
25%	103
50%	104
75%	105
max	127

Figure 5.14: PublicBIbenchmark IGlocations2 url Column Results



Row	Value
1	/r/leagueoflegends/comments/ab8wps/whats_the_bestfunniest_insults_youve_ever_heard/eczayvu/
2	/r/GlobalOffensive/comments/aaz48j/thorins_top_20_csgo_storylines_of_2018_part_1_2011/eczaz05/
3	/r/hiphopheads/comments/ab966a/machine_gun_kelly_reignites_eminem_beef_fuck_rap/eczaz80/
4	/r/GlobalOffensive/comments/abcrwy/alttab/eczaz8u/
5	/r/leagueoflegends/comments/aba6t3/i_played_over_1000_games_of_aram_in_season_8_and/eczazgd/
6	/r/Android/comments/ab37f1/whats_the_point_of_having_phones_with_8gb_to_10gb/eczazgj/
7	/r/MechanicalKeyboards/comments/abcrxu/received_the_wrong_switches_from/eczazgt/
8	/r/GlobalOffensive/comments/abatly/i_smoked_window_from_apps_and_im_proud/eczazpe/
9	/r/leagueoflegends/comments/abcmhk/so_no_your_year_in_review_for_this_year/eczazq9/
10	/r/CalPoly/comments/aa3icz/coop_and_federal_loan_grace_period/eczazwg/
11	/r/leagueoflegends/comments/ab8wps/whats_the_bestfunniest_insults_youve_ever_heard/eczazxc/
12	/r/GlobalOffensive/comments/abcobz/tarik_deleted_twitlonger/eczb04k/
13	/r/GlobalOffensive/comments/abatly/i_smoked_window_from_apps_and_im_proud/eczb05m/
14	/r/hiphopheads/comments/ab8k8w/fire_in_the_booth_migos/eczb09s/
15	/r/hiphopheads/comments/ab8ns8/layzie_bone_admits_his_feelings_are_hurt_it_hurts/eczb0ag/
16	/r/leagueoflegends/comments/ab8woo/best_skin_of_2018/eczb0fm/
17	/r/indieheads/comments/ab33h1/underrated_albums_of_2018_as_determined_by_you/eczb0o6/
18	/r/Android/comments/abaq93/okay_google_what_happened_linus_tech_tips_pixel/eczb0sy/
19	/r/leagueoflegends/comments/ab7t0f/banned_for_14_days_due_to_mistaken_identity/eczb0ti/
20	/r/Android/comments/abaq93/okay_google_what_happened_linus_tech_tips_pixel/eczb0z2/
21	/r/leagueoflegends/comments/aat6bf/new_player_looking_for_worst_character/eczb18s/
22	/r/Android/comments/abaq93/okay_google_what_happened_linus_tech_tips_pixel/eczb18v/
23	/r/leagueoflegends/comments/ab8wps/whats_the_bestfunniest_insults_youve_ever_heard/eczb1c5/
24	/r/Android/comments/ab2dh8/beware_of_port_out_scam_scary_threat_to_your/eczb1c9/
25	/r/malefashionadvice/comments/abcqoe/is_this_ok/eczb1ev/

Statistic	Value
count	100000
mean	80.70309
std	11.05123
min	35
25%	74
50%	83
75%	90
max	98

Figure 5.15: NextiaJD Reddit_Comments_7M_2019 permalink Column Results

5.3.4 Aggregated Analysis

Let us analyze all the results above, aggregated in a box plot, where the dotted line shows us the mean:

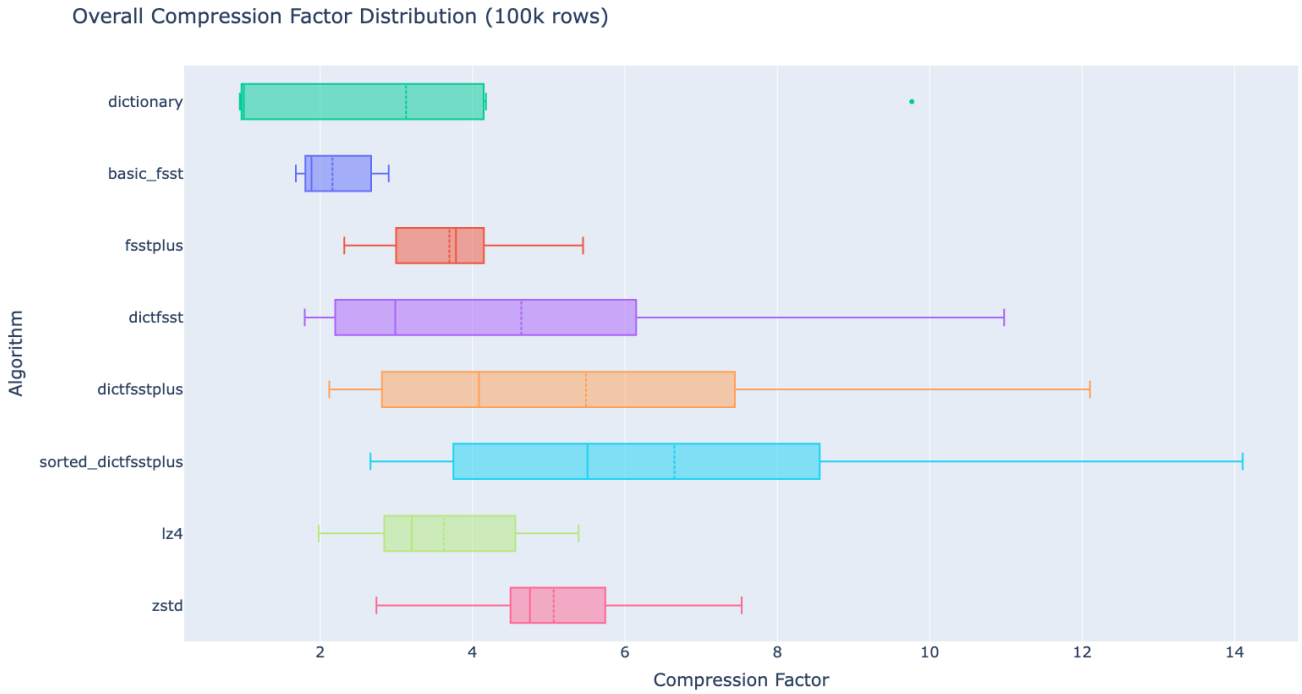


Figure 5.16: Compression Factor Box Plot - Dynamic Programming

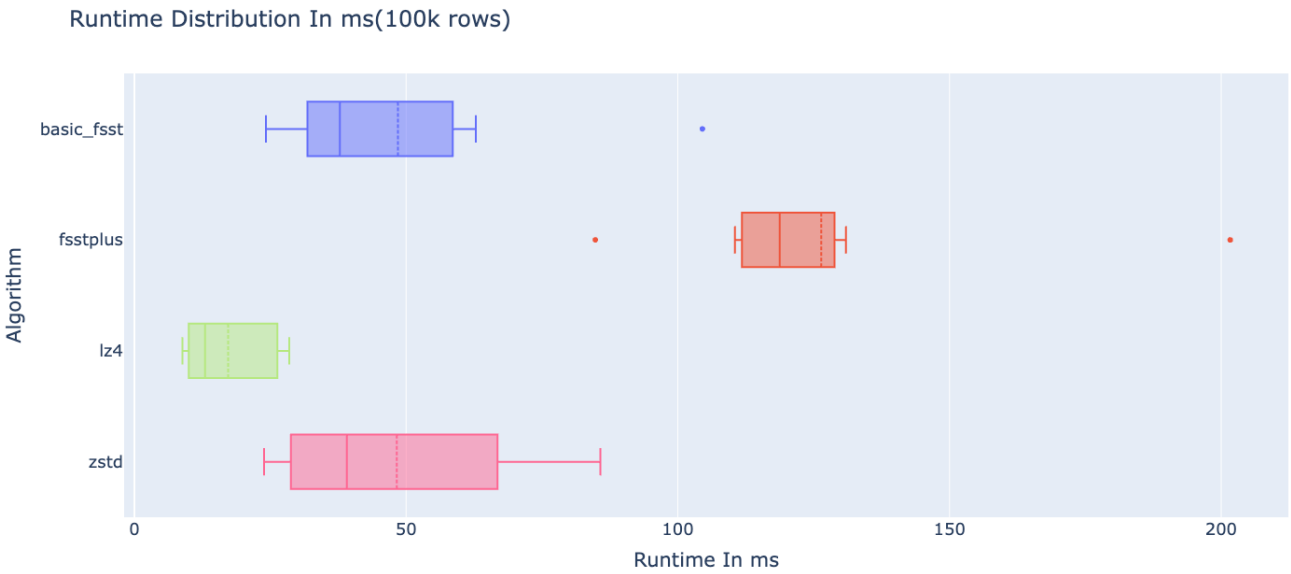


Figure 5.17: Compression Runtime Box Plot - Dynamic Programming

Below are the documented figures measured. (For dictionary variants, the compression ratios were theoretically measured by using DuckDB and SQL to pre-process the dataset, keep only distinct values, and run the algorithms on that; therefore, no runtime was measured.)

Table 5.2: Dynamic Programming Performance On Prefix-Rich Datasets

Algorithm	**Mean Compression Factor**	**Mean Runtime (ms)**
basic_fsst	2.16	48.49
fsstplus	3.7	126.43
dictfsst	4.64	-
dictfsstplus	5.49	-
sorted_dictf-sstplus	6.65	-
lz4	3.62	17.26
zstd	5.07	48.26

On the selected prefix-rich datasets, the analysis reveals that:

- Compared to Basic FSST, FSST+ provides on average:

  - A 70% improvement in compression factor
  - With a 2.6x slower compression time

- DICT FSST+ provides on average:
  - Compared to FSST
    - A 155% increase in compression factor
    - Potentially reaching 207% by sorting the dictionary beforehand
  - Compared to DICT FSST:
    - A 19% increase in compression factor
    - Potentially reaching 43% by sorting the dictionary beforehand
  - Compared to Zstandard:
    - An 11% increase in compression factor
    - Potentially reaching 31% by sorting the dictionary beforehand

These results demonstrate that the dynamic programming solution is well-suited for finding ideal prefixes in a corpus within a reasonable time, outperforming LZ4's compression ratio on average for this class of data. Furthermore, DICT FSST+ and its sorted variant show remarkable compression factor potential, with the sorted version consistently beating zstd on average for these prefix-rich datasets.

5.3.5 Aggregated Full DICT FSST+ Analysis

It is also interesting to observe how DICT FSST+, and especially Sorted DICT FSST+, behave on all columns without selecting for prefix-richness. The maximum number of rows for this benchmark was limited to 100k.

Here is a box plot for the compression factor on all columns where FSST's compression factor is twice that of dictionary encoding. There are 246 of those.

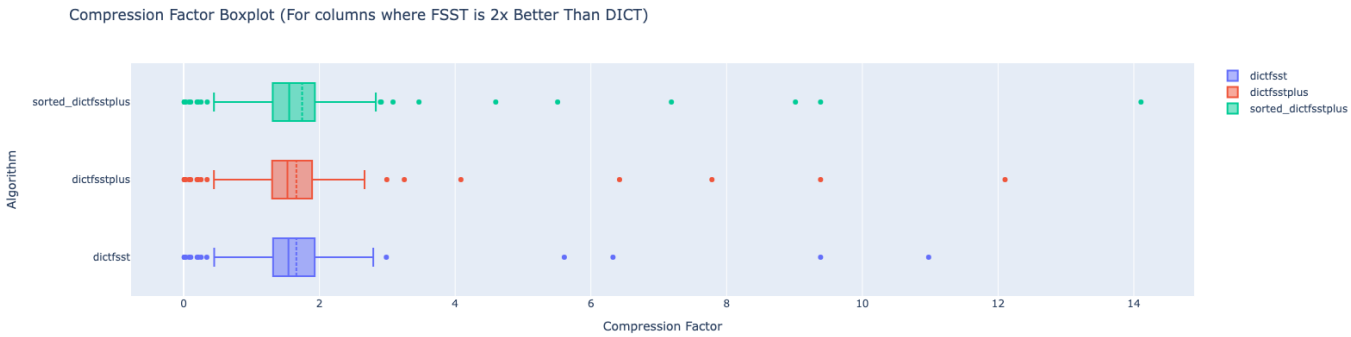


Figure 5.18: Compression Factor Box Plot For Columns Where FSST Is 2x Better Than Dict

We observe several outliers, with the highest value achieved by Sorted DICT FSST+. Zooming in:

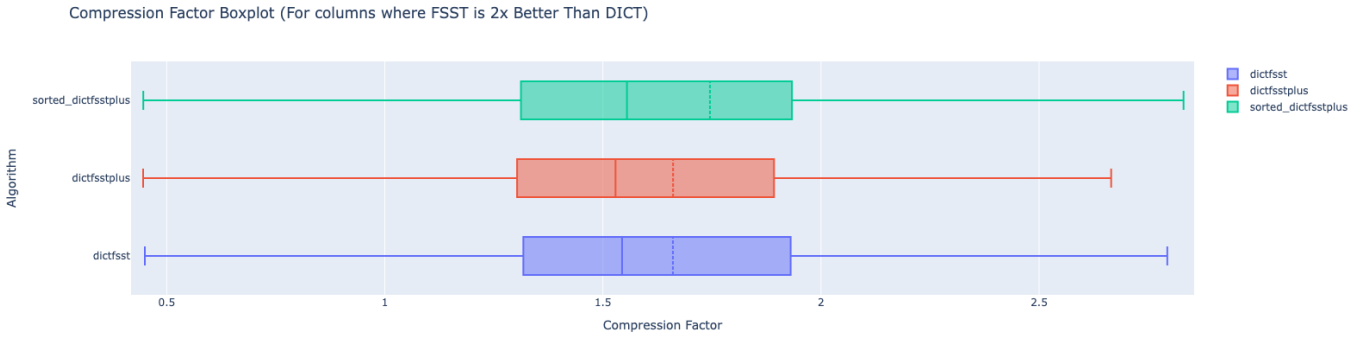


Figure 5.19: Compression Factor Box Plot For Columns Where FSST Is 2x Better Than Dict Zoomed In

We observe a clear gain in the mean compression factor (the dotted line) for Sorted DICT FSST+ compared to its non-sorted counterpart and to DICT FSST. What we can also notice is that the median remains relatively unchanged, while the mean is higher. That means a small amount of results is skewing the mean, which makes sense, since not every dataset has prefixes so that most datasets will show no gain.

Now, here is the box plot for all columns where the previous filter condition does not hold, i.e., where FSST's compression factor is not twice that of dictionary encoding. There are 599 of those.

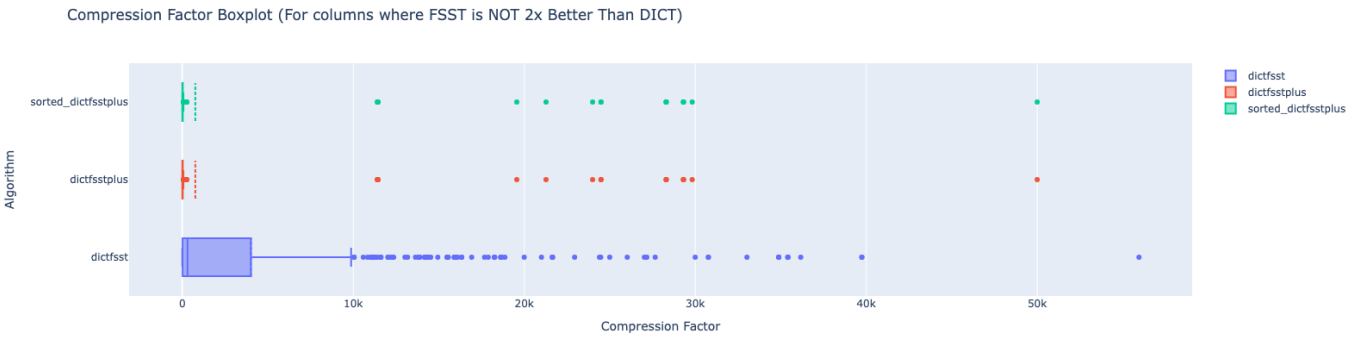


Figure 5.20: Compression Factor Boxplot For Columns Where FSST Is NOT 2x Better Than Dict

Here we see very extreme compression factor values, which make sense since dictionary encoding performs well in these columns. The mean compression factor of DICT FSST here is 4015, whereas for both DICT FSST+ and Sorted DICT FSST+, it is 766. Among these columns, there is not much gain to be had for DICT FSST+ and its sorted variant.

5.4 Experiments

This section documents additional experiments conducted to achieve the solution presented above.

5.4.1 Different Block Sizes

Before deciding to stick with the block size value of 128, different values were experimented with, running benchmarks with block sizes of 64, 128, and 192. Here are the results (among prefix-rich columns) with a block size of 64 elements (the dotted line showing the mean):

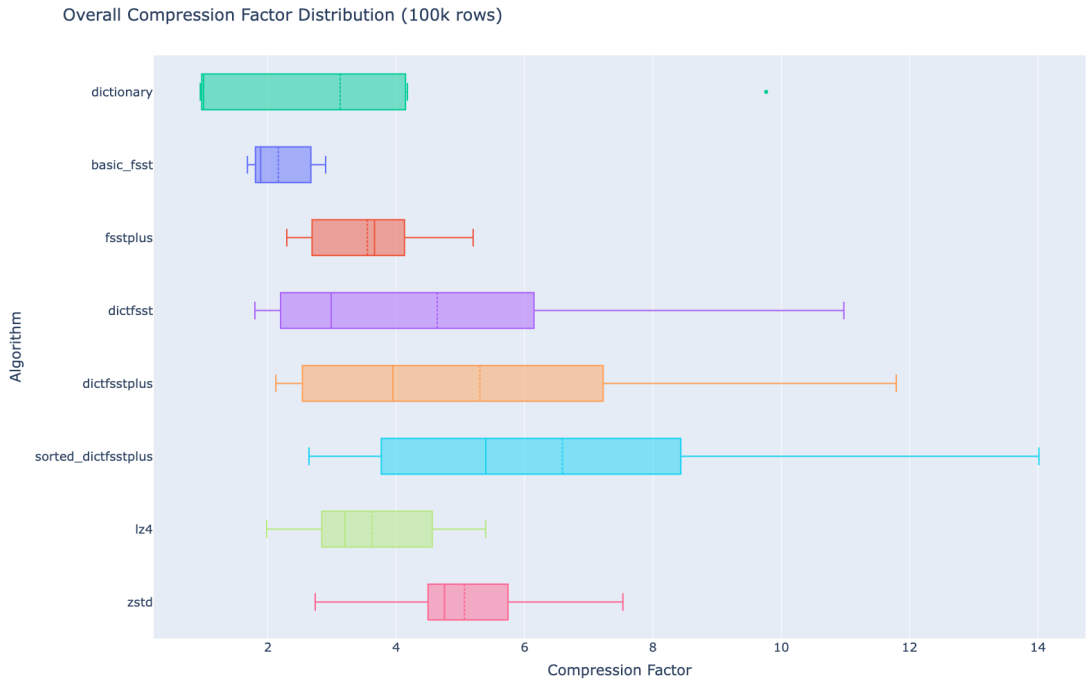


Figure 5.21: Blocksize 64 Compression Factor Box Plot

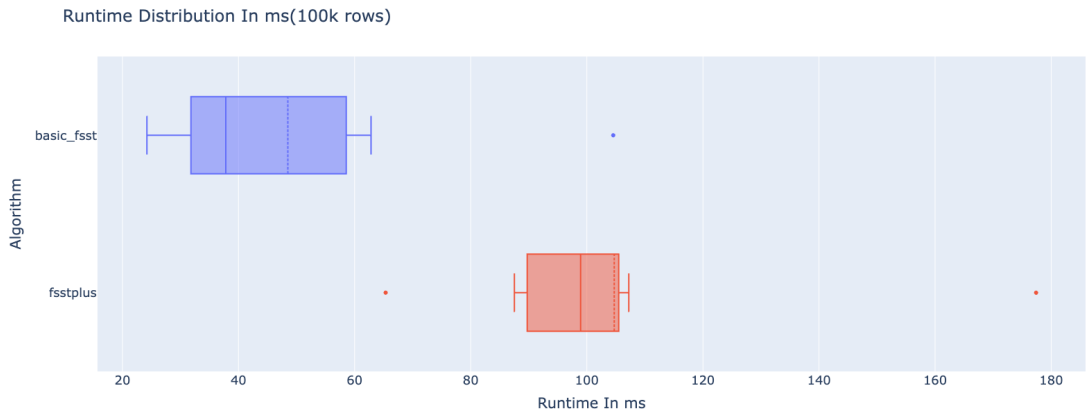


Figure 5.22: Blocksize 64 Compression Runtime Box Plot

Here are the results with a block size of 192:

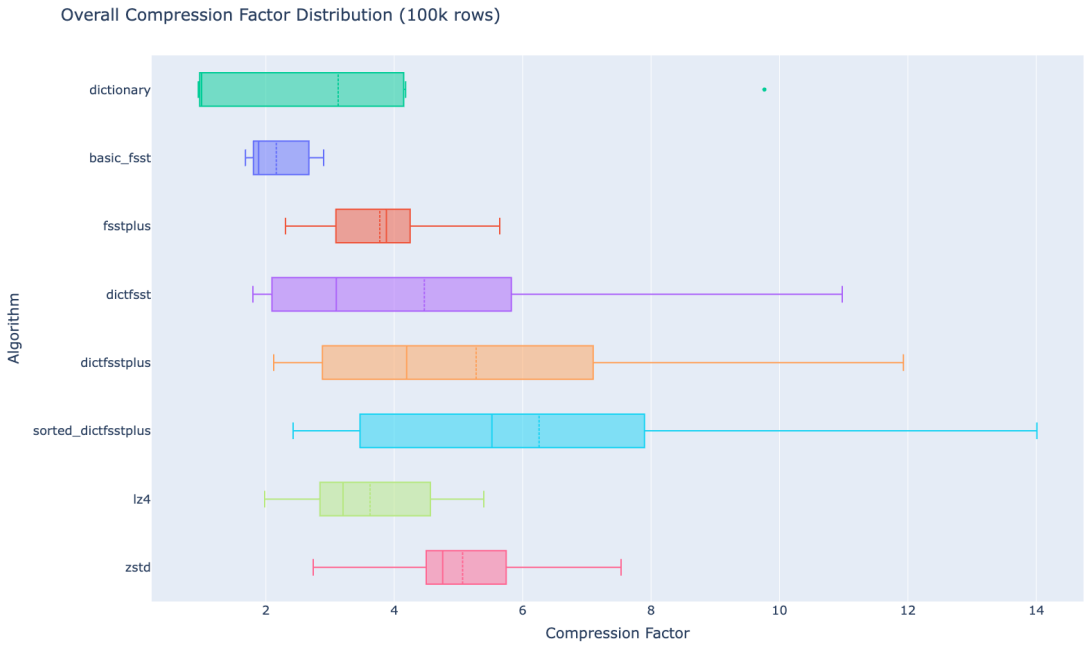


Figure 5.23: Blocksize 192 Compression Factor Box Plot

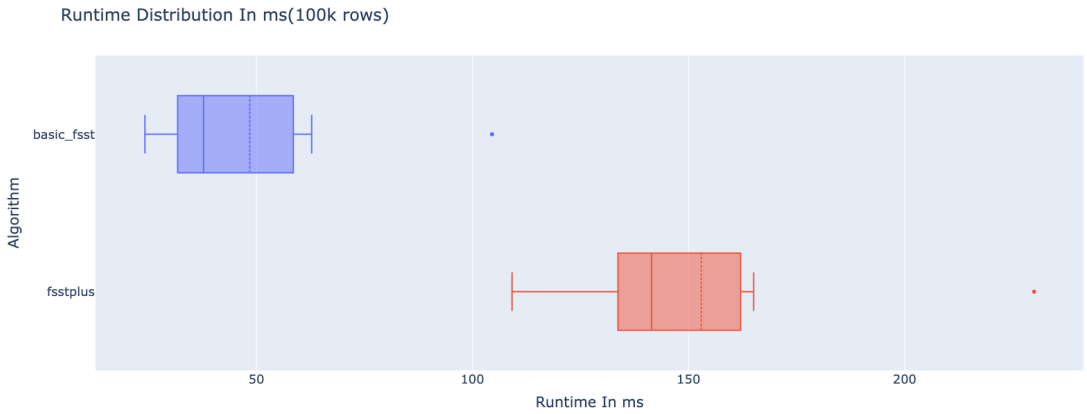


Figure 5.24: Blocksize 192 Compression Runtime Box Plot

Summarizing the results in a table:

Table 5.3: Performance comparison of different block size values

Algorithm	**Mean Compression Factor**	**Mean Runtime (ms)**	**Efficiency (Compression Factor Divided By Runtime)**
basic_fsst	2.16	48.49	0.04454526707
fsstplus blocksize64	3.55	104.72	0.03389992361
fsstplus blocksize128	3.70	126.43	0.02926520604
fsstplus blocksize192	3.78	152.99	0.02470749722

By plotting this data, we get the following:

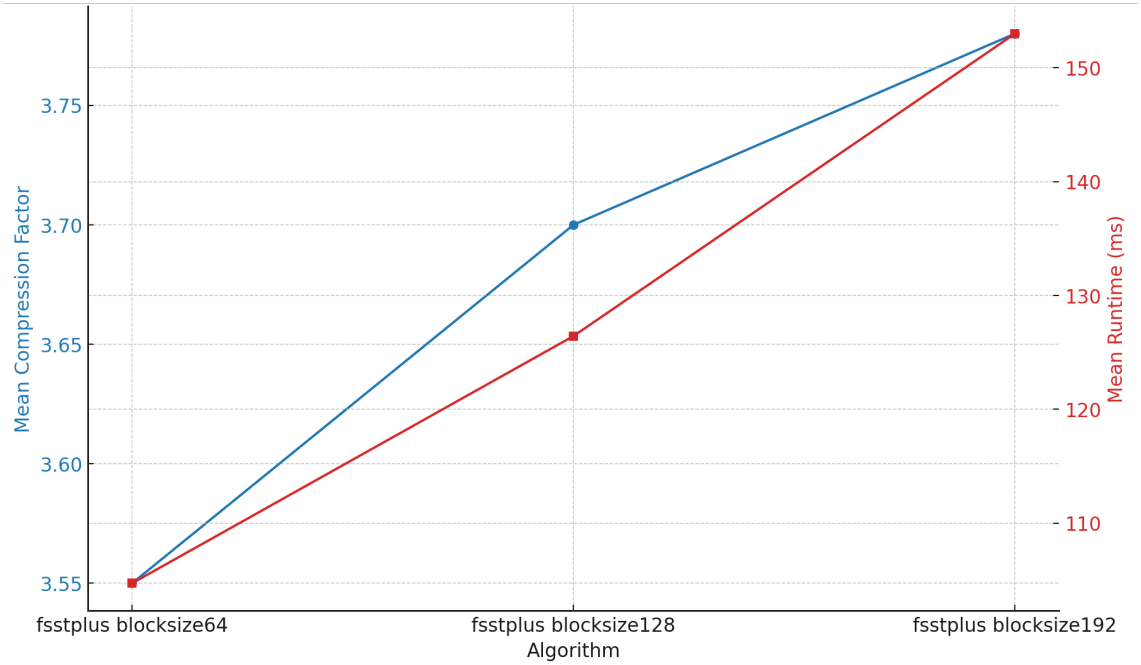


Figure 5.25: Compression Factor and Runtime per Blocksize

If we observe the general trend of the plot, we see that the compression factor approaches a plateau as the block sizes increase. At the same time, the runtime continues to scale, increasing in a slightly accelerating fashion. This makes sense, as the dynamic programming

algorithm runs per block and scales quadratically. Therefore, by increasing the block size, the runtime should continue to increase at a faster rate until reaching a block size of n (with all elements), resulting in a quadratic time complexity across all elements.

As we can see, the most cost-effective bang-for-your-buck approach out of the three tested is block size 64, with an efficiency value of 0.0339; however, in this case, we don't achieve as high a compression factor.

Looking at the other extreme, with a block size of 192, we get diminishing returns, as shown by a sharp drop in efficiency, down to 0.0247.

The trade-off of a slightly increased runtime for a significantly improved compression factor led to the choice of a balanced configuration with 128 elements per block. The main advantage of FSST+ is its high compression factor for specific datasets, which is where the emphasis lies. This setting achieves an acceptable efficiency of 0.029, a high mean compression ratio, and a runtime approximately 2.6× that of `basic_fsst`.

It is important to note that this block size is a tunable parameter. Practitioners applying the algorithm in different contexts may choose to adjust this value to better suit their specific performance or compression requirements, depending on the characteristics of their data and workload constraints.

5.4.2 Two Symbol Tables, Pre-Compression, and Longer Prefixes

A key design decision in FSST+ is how to apply FSST compression to the cleaved prefixes and suffixes. One approach is to use two separate symbol tables, one trained specifically on the set of prefixes and another on the set of suffixes. This could theoretically improve compression by creating more specialized tables. The alternative is to use a single, unified symbol table trained on a sample of the combined corpus of prefixes and suffixes.

To evaluate this trade-off, both approaches were benchmarked and compared. The two-symbol-table variant did not yield significant benefits. Across all benchmark columns, the largest compression factor gain was a mere 3%. At the same time, it offered no improvement in most datasets and, in some cases, resulted in a slightly worse compression ratio (a decrease of approximately 2%). The average compression factor across prefix-rich datasets for the two-symbol-table variant was 3.694, compared to 3.696 for the single-table variant. Given these negligible gains, the single-table approach was selected as the default. It avoids the complexity and overhead of creating, managing, and storing a second symbol table with the compressed data.

This section further documents findings from related experiments exploring this design choice, alongside other optimizations:

- `fsstplus_onest`
  - Using one symbol table for compressing both prefix and suffix after determining the prefixes.
- `fsstplus_twost`
  - Using two symbol tables for compressing prefix and suffix separately.
- `fsstplus_onest_compressbefore`
  - Using one symbol table for compressing the entire strings right away, and then determining prefixes by running the dynamic programming algorithm on the already compressed data. This is the variant used in Section 5.3.
- `fsstplus_longerprefix_onest_compressbefore`
  - A variant of the previous algorithm, but in this case, suffixes in a chunk can use different lengths of a single prefix, sometimes re-using the same prefix instead of writing a new one. This makes each prefix that is written more valuable, as we would not need to write a new prefix if it is a sub-string of an already written prefix.
- `fsstplus_longerprefixnocmp_onest_compressbefore`
  - A variant of the previously described algorithm but with no early pruning attempts.

The variants above test two fundamentally different strategies. The first, exemplified by `fsstplus_onest` and `fsstplus_twost`, involves first identifying prefixes in the raw data and then compressing the resulting prefixes and suffixes. A more advanced strategy, used in `fsstplus_onest_compressbefore`, is to apply FSST compression to the entire dataset at the very beginning and then run the dynamic programming algorithm on the already-compressed byte streams.

This ”compress-first” approach introduces several important trade-offs. On one hand, it simplifies the pipeline by eliminating the Cleaving step, which is no longer needed to separate data for the FSST API. More significantly, it allows for much longer effective prefixes. When working with raw strings, a prefix length is limited to 120 bytes to avoid its worst-case FSST-compressed representation exceeding the 255-byte limit of the `uint8_t`

length field. By operating on pre-compressed data, we can define prefixes up to 255 compressed bytes long, capturing much larger redundancies.

On the other hand, this method adds complexity to the dynamic programming stage. The LCP calculation must be careful not to split FSST's escape code (byte 255) from its subsequent literal byte, as this would corrupt the meaning of the compressed stream. Furthermore, the granularity of prefix matching is reduced. An optimal prefix in the original data might end in the middle of an 8-byte FSST symbol; in such a case, the algorithm will subtract 1 byte from the prefix, making it shorter and aligning with the symbol boundary. This can lead to slightly less optimal chunking. The following results explore the practical impact of these design choices.

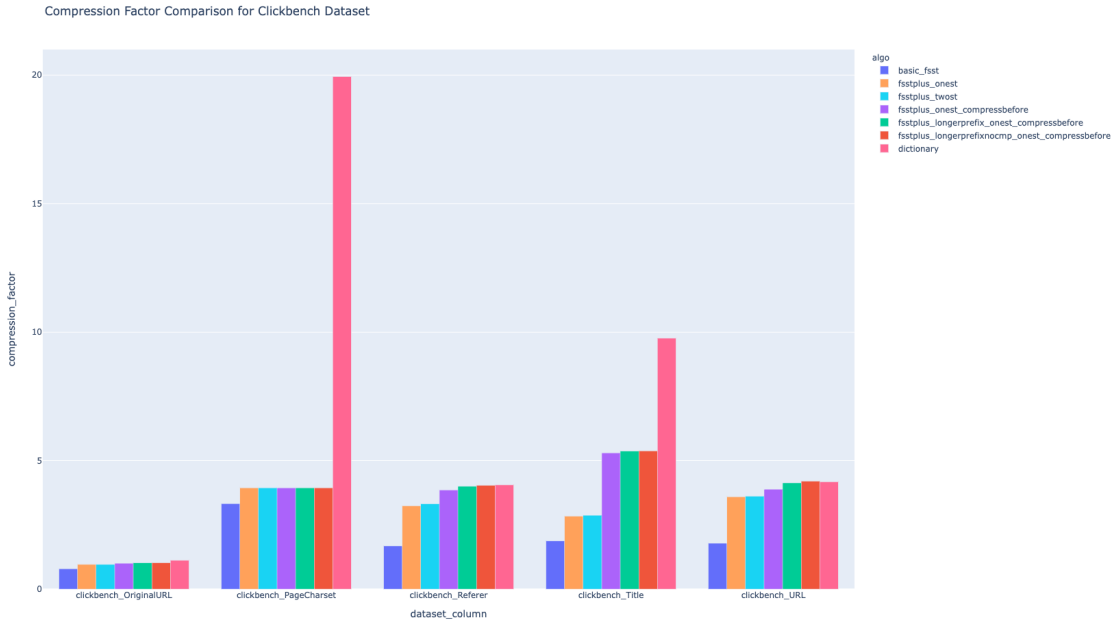


Figure 5.26: Clickbench Columns Experiments Results

Here we see that the `fsstplus_longerprefix_onest_compressbefore` variant is able to achieve a higher compression ratio of 4.13 for the URL column, better than the previous best result. Unfortunately, it takes orders of magnitude longer to run, roughly 20 times the runtime of Basic FSST, making it infeasible for practical use. The `fsstplus_longerprefixnocmp_onest_compressbefore` variant, which has no early pruning, shows a slightly higher compression factor for URL but is many more orders of magnitude slower and thus out of consideration. We also see a significant improvement for the 'Title' column with `fsstplus_onest_compressbefore`. This is a direct consequence

of the ”compress-first” strategy. As explained, it allows for a maximum prefix size of 255 compressed bytes. This enables the algorithm to capture the very long repeated phrases common in the ‘Title‘ data, a capability that would be impossible if we were limited by the theoretical worst-case expansion of uncompressed data.

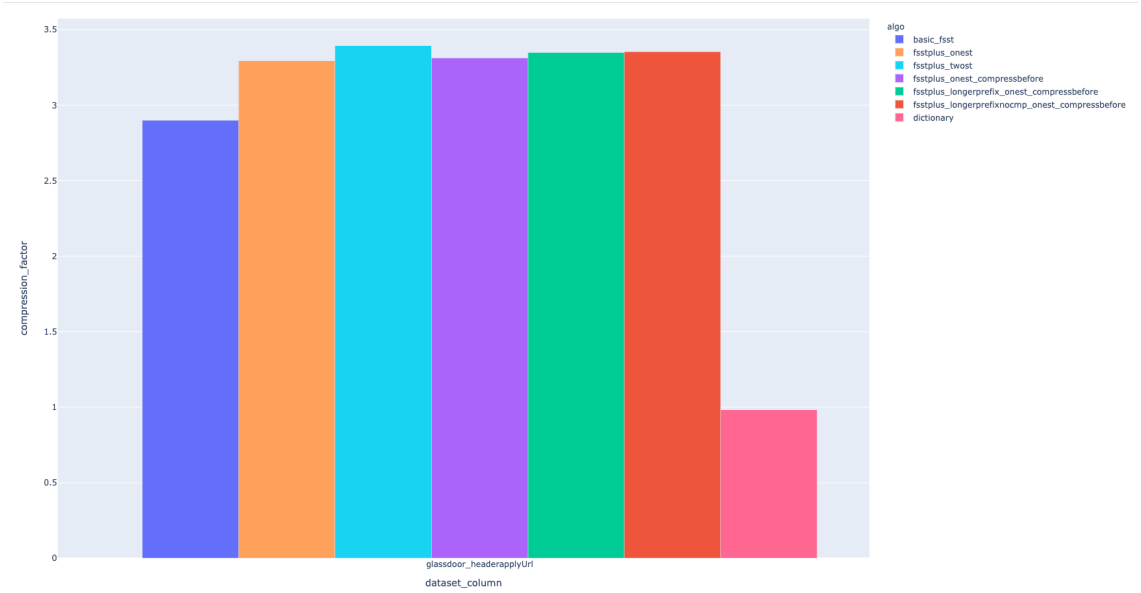


Figure 5.27: NextiaJD Glassdoor headerapplyUrl column experiments results

Here we see that `fsstplus_twost` shows a slight improvement in compression factor over its single-table counterpart. However, the gain is not substantial enough to warrant using it instead of the final selected variant, `fsstplus_onest_compressbefore`, which remains fast and provides great compression factor results.

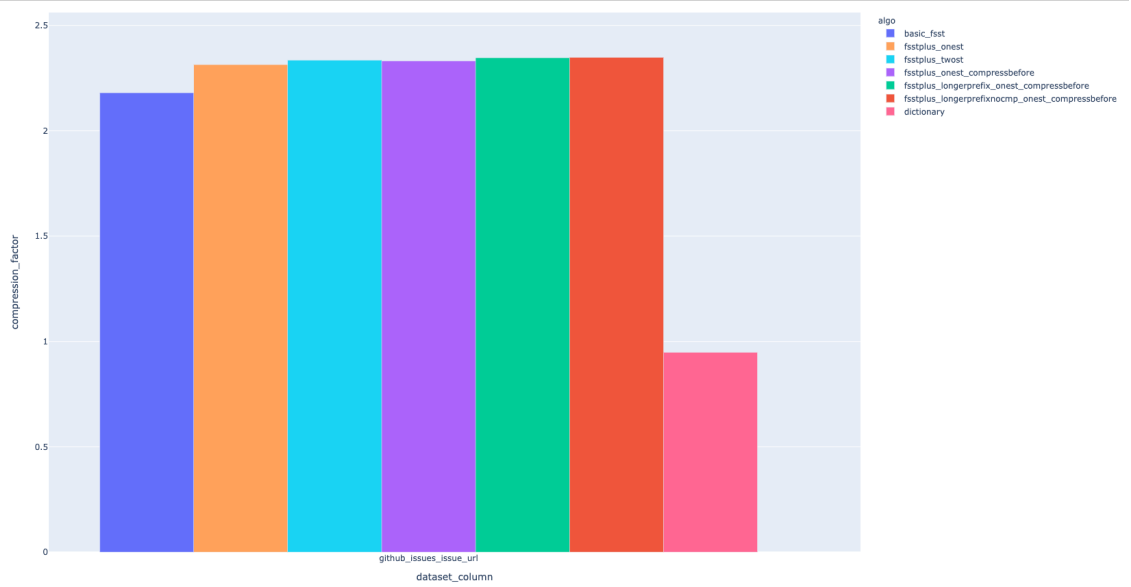


Figure 5.28: NextiaJD github_issues issue_url column experiments results

For this column, there is mostly no gain or loss, even for the `fsstplus_longerprefixnocmp_onest_compressbefore` variant, showing that the orders of magnitude higher runtime tradeoff is not worth it once again.

5.4.3 Recursive FSST

Before committing to the more complex dynamic programming solution, a simpler, more direct method for potentially improving compression was explored: Recursive FSST. This approach investigates whether iteratively applying the FSST algorithm to its own output can yield further compression gains. The core idea is to first compress the original string corpus using a standard FSST pass, which generates a compressed byte stream and a corresponding symbol table. Then, this resulting byte stream is treated as a new dataset and is compressed again with a second, independently trained symbol table. This process can be repeated for multiple passes. This experiment, while not directly related to the dynamic programming or prefix-sharing logic, serves as an important baseline. It helps to determine whether the residual patterns left in an FSST-compressed stream can be further compacted by the same algorithm, or if a fundamentally different approach, such as FSST+, is required to exploit other types of redundancy. To ensure a fair comparison, the final compressed size for this recursive method must include the storage overhead of all

symbol tables generated, as each one is required to reverse the multi-stage compression process. This experiment was conducted on the URL, Referer, and Title columns from the ClickBench dataset, with the results for up to four recursive passes shown in Table 5.4.

Table 5.4: Compression Factor Comparison for Recursive FSST

Column	FSST	2xFSST	3xFSST	4xFSST	FSST+
clickbench.URL	1.79	2.03	2.09	2.09	3.78
clickbench.Referer	1.68	1.84	1.87	1.87	3.32
clickbench.Title	1.87	1.94	1.95	1.95	4.29

The results clearly demonstrate a pattern of rapidly diminishing returns. While a second pass (2x FSST) offers a marginal improvement over a single pass, subsequent passes yield almost no additional benefit. For the clickbench.In the URL column, the compression factor increases from 1.79 to 2.03 in the second pass, but only rises to 2.09 in the third pass before plateauing completely.

These initial results might be influenced by the high number of duplicate strings in the original clickbench datasets. For instance, in the clickbench.URL column, 81.82% of the values are duplicates. This raises an interesting question: how does recursive compression perform on a corpus of unique strings, where redundancy exists purely at the substring level? To investigate this, we conducted a second experiment where we applied the same recursive FSST technique to the dictionary of unique strings from each column, as described in Section 4.2.1. The results are presented in Table 5.5.

Table 5.5: Compression Factor Comparison for DICT Recursive FSST

Column	DICT FSST	DICT 2xFSST	DICT 3xFSST	DICT 4xFSST	DICT FSST+	Sorted DICT FSST+
clickbench.URL	6.33	6.96	7.09	7.13	7.78	9.01
clickbench.Referer	5.61	6.03	6.10	6.11	6.42	7.19
clickbench.Title	10.98	11.19	11.17	11.16	12.10	14.11

The findings from this dictionary-based experiment are similar to the previous one. Applying a second pass (DICT 2x FSST) provides a noticeable improvement over a single pass, but subsequent passes again show rapidly diminishing returns. For clickbench.URL, the compression factor improves from 6.33 to 6.96, but only reaches 7.13 after four passes. Interestingly, for the Title column, the compression factor slightly decreases after the second pass. To investigate further why that is, we can first take a look at the symbol table sizes. For the DICT 4x FSST run, the symbol table sizes were respectively 543, 284,

272 and 272 bytes, together totaling 1,371 bytes, which is proportionally insignificant to the total compressed size of 2,017,140 bytes for that run. If we recalculate the compression factors, but this time ignore the overhead of the symbol table sizes for the Title column, we get for DICT 2xFSST 11.19, for 3x 11.18 and for 4x 11.17. This leads us to the conclusion that more than two runs of recursive FSST compression do not yield significant improvements, not because of the symbol table overhead but because of the inner workings of the algorithm itself.

While recursive dictionary compression shows some potential for slightly more compression, it consistently underperforms compared to the targeted approach of DICT FSST+. In every case, DICT FSST+ achieves a higher compression factor than even four passes of recursive FSST. Furthermore, the Sorted DICT FSST+ variant, which leverages lexicographical ordering to maximize prefix sharing, significantly outperforms all other methods. These results show that for prefix-heavy data, a purpose-built algorithm like FSST+ can reach higher compression ratios than simply repeatedly applying FSST to its output.

However, it is important to recognize that the two approaches target different kinds of redundancy. FSST+ is highly specialized, designed to find and eliminate shared prefixes. A recursive FSST application, in contrast, is a generalist; in theory, it could identify any frequent substring pattern remaining in the compressed stream, including those in the middle or at the end of strings—areas that a prefix-only model would inherently miss. This suggests a potential avenue for future research, though the trade-offs in performance and the overhead of managing multiple symbol tables would need to be carefully evaluated.

6

Hash Grouping Solution

To explore an alternative to the sort-based dynamic programming approach, a heuristic method based on hash grouping was developed. The goal of this approach was to avoid the O(n log n) sorting cost per batch by directly grouping strings with identical prefixes. This chapter details the algorithm and its performance, ultimately demonstrating why the DP-based approach remains superior.

The idea for this approach is that, by using hashing, we could avoid first have to sort the strings, but can group them together if their prefixes hash to the same value, and at the same time use a cost function to find good prefixes.

6.1 Group Data Structure

Before diving into the hash grouping algorithm, we need to understand the core data structure used: the `Group` class. A Group represents a collection of strings that share a common prefix of a certain length. This Group structure is now used, instead of the "Similarity Chunks" from the dynamic programming solution.

```
class Group {
    std::vector<uint8_t> indices;        // Local indices (0-127) of strings in
    ↪  this group
    bool can_go_further;                 // Whether we can extend the prefix
    ↪ further
    uint8_t horizontal_offset;           // Current prefix length being
    ↪ considered

Public:
    const int PREFIX_REFERENCE_COST_PER_ELEMENT = 2;

    // Calculate compression gain from using this prefix length
```

```
    int calculate_gain() const {
        if (indices.size() <= 1) return 0;

        size_t n_elements_profiting = indices.size() - 1;
        int gain = n_elements_profiting * horizontal_offset;
        gain -= indices.size() * PREFIX_REFERENCE_COST_PER_ELEMENT;
        return gain;
    }

    bool has_prefix() const {
        return indices.size() > 1 && horizontal_offset > 0;
    }
};
```

Listing 6.1: Group Class Structure

The key insight is that a Group with `horizontal_offset = k` means all strings in this group share the same first `k` bytes. The `calculate_gain()` function computes the compression benefit: we save `k` bytes for each string except one (which stores the actual prefix), but pay the overhead cost of storing references to the prefixes (jump-back offset). To support the new Group data structure, the Sizing and Writing code was restructured and algorithmically refined, though we won't delve into the details here.

6.2 Hash Grouping Algorithm

The hash grouping algorithm provides an alternative to the dynamic programming approach that avoids the need for sorting. Instead of lexicographically ordering strings and then finding optimal split points, it uses hashing to group strings with similar prefixes and employs a recursive splitting strategy.

6.2.1 Algorithm Overview

Let's take a look at the following Python Pseudocode:

```
def hash_grouping_algorithm(strings):
    # Start with all strings in one group
    initial_group = Group(
        indices=list(range(len(strings))),
        can_go_further=True,
        horizontal_offset=0
    )

    # Recursively split to find optimal grouping
    optimal_groups = split_recursive(initial_group, strings)
```

```
    return optimal_groups

def split_recursive(group, strings):
    # Base cases: stop recursion
    if len(group.indices) == 1 or not group.can_go_further:
        return [group]

    # Current best solution: keep group as-is
    selected_groups = [group]

    # Try subdividing this group
    subdivided_groups = subdivide_group(group, strings)

    # Recursively split each subdivided group
    for sub_group in subdivided_groups:
        recursive_splits = split_recursive(sub_group, strings)

        # For each split, try merging it with our current selection
        for split_group in recursive_splits:
            new_groups = merge_group(selected_groups, split_group)

            # Keep the better option based on compression gain
            if calculate_total_gain(new_groups) > calculate_total_gain(
    ↪ selected_groups):
                selected_groups = new_groups

    return selected_groups

def subdivide_group(group, strings):
    # Hash strings based on 8-byte chunk at current offset
    hash_map = {}
    offset = group.horizontal_offset

    For the index in the group.indices:
        string = strings[index]

        if len(string) >= offset + 8:
            # Hash 8-byte chunk starting at offset
            chunk = string[offset:offset+8]
            hash_value = hash(chunk)
        else:
            # Use remaining length as distinguishing feature
            hash_value = len(string) - offset

        if hash_value not in hash_map:
            hash_map[hash_value] = []
        hash_map[hash_value].append(index)
```

```
57
58      # Create new groups based on hash buckets
59      new_groups = []
60      for hash_value, indices in hash_map.items():
61          can_continue = all(len(strings[i]) >= offset + 8 for i in indices)
62          next_offset = offset + 8 if can_continue else offset
63
64          new_group = Group(
65              indices=indices,
66              can_go_further=can_continue,
67              horizontal_offset=next_offset
68          )
69          new_groups.append(new_group)
70
71      return new_groups
72
73  def merge_group(selected_groups, new_group):
74      # Remove indices that overlap with new_group from selected_groups
75      # and add the new_group
76      new_indices_set = set(new_group.indices)
77      result = []
78
79      for group in selected_groups:
80          filtered_indices = [i for i in group.indices if i not in
    ↪ new_indices_set]
81          if filtered_indices:  # Only keep non-empty groups
82              result.append(Group(filtered_indices, group.can_go_further, group.
    ↪ horizontal_offset))
83
84      result.append(new_group)
85      return result
```

Listing 6.2: Hash Grouping Algorithm (Pseudocode)

This approach eliminates the initial O(n log n) sorting step for each batch done for dynamic programming, which can be significant for large datasets. However, the trade-off is that this approach does not always find the globally optimal solution that dynamic programming guarantees, as it employs a greedy algorithm in the merging step.

It will be executed on runs of 128 strings, the same as the dynamic programming approach of Chapter 5. By keeping it to runs of 128 strings, the data structures used to track, for instance, hash keys used so far, are kept within a constrained range.

6.2.2 Key Algorithmic Components

1. Hashing Strategy: Instead of comparing strings character by character, the algorithm hashes 8-byte chunks at the current horizontal offset. This provides several advantages:

- **Speed:** Hashing is faster than byte-by-byte comparison, especially for longer strings
- **Granularity:** Processing 8 bytes at a time aligns well with modern CPU architectures
- **Collision Handling:** The linear probing hash table efficiently handles collisions while maintaining cache locality

2. Recursive Splitting: The algorithm explores the space of possible groupings by recursively subdividing groups. At each level, it:

- Maintains the current best grouping (`selected_groups`)
- Tries subdividing the current group based on the next 8-byte chunk
- For each subdivision, it recursively explores further splits
- Uses a greedy selection strategy, keeping improvements that increase total compression gain

3. Cost Function Integration: The `calculate_gain()` function from the Group class provides the optimization criterion. The algorithm continuously evaluates whether a new grouping configuration provides better compression than the current one.

6.3 Results

6.3.1 Per-column Analysis

The results show the compression factor on the same prefix-rich columns selected at Section 5.3.2, with the following algorithms :

- Basic FSST
- FSST+ (With Hash Grouping)
- Dictionary Encoding
- DICT FSST (Compressing the dictionary with FSST)
- DICT FSST+ (Compressing the dictionary with FSST+)

- Zstd (with 64kb block size)
- Lz4 (with 64kb block size)

The same bit-packing considerations mentioned at Section 5.3.2 apply.

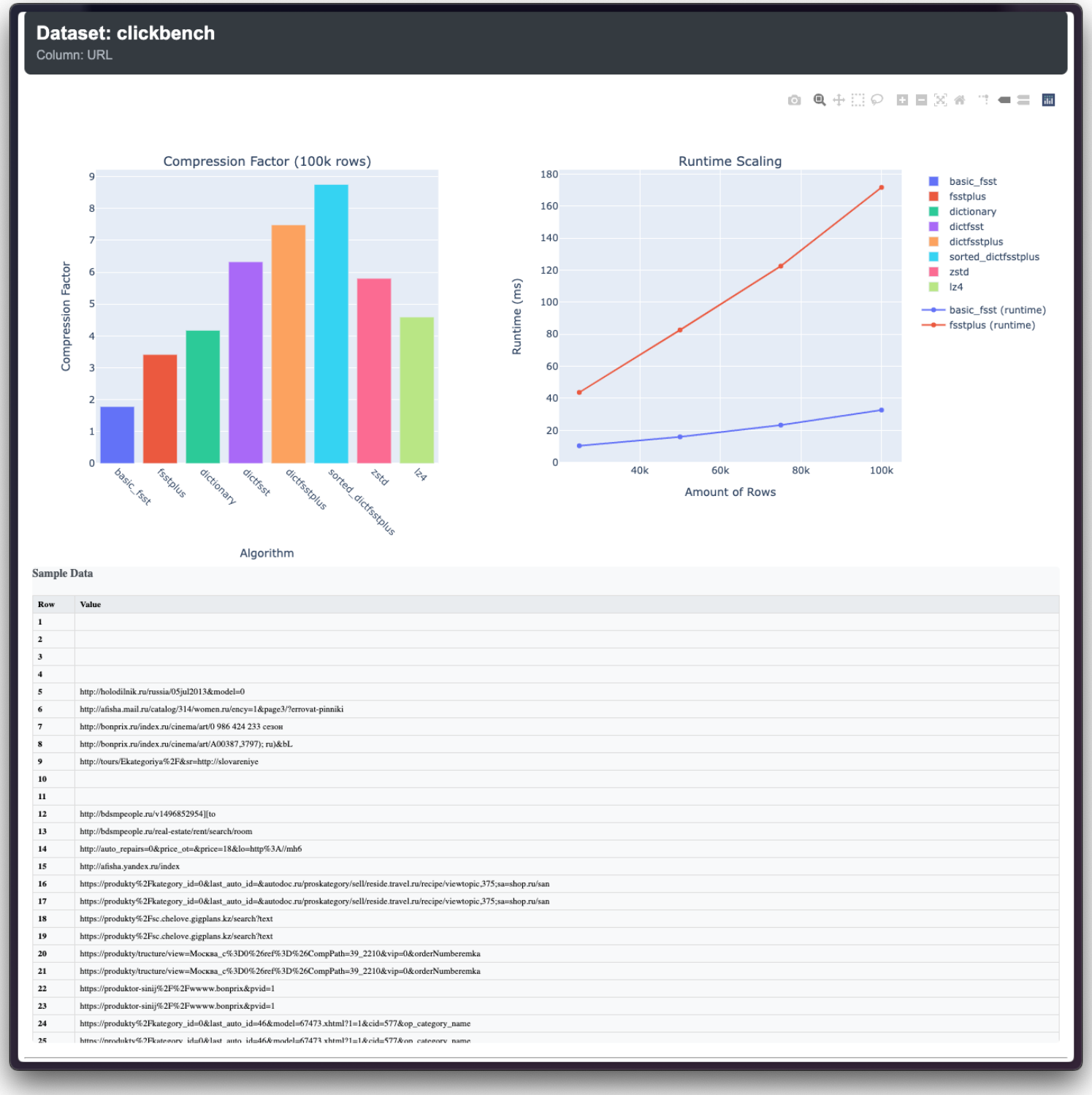


Figure 6.1: ClickBench URL Column Results

Row	Value
37	Легко на участные участников., Цены - Стильная парнем. Саганрог догадения : Турции, купить у 10 дне кольные машинки не представки - Новая с избиение спродажа: котята 2014 г.в. Цена: 47500-10ECO060 – -...
38	Легко на участные участников., автосалоны, купить женщину LG в Минск, транспорядка на Бибика.ру - Автопоиск по низкой обладам - Человечерный. Есть в интернет-магазине легко нату. Так соток,
39	Легко на участные участников., автосалоны, купить женщину LG в Минск, транспорядка на Бибика.ру - Автопоиск по низкой обладам - Человечерный. Есть в интернет-магазине легко нату. Так соток,
40	Легко на MyLove.Ru / Наконическая одежды и клиника, Омске вариум, центры «Единство футфетиш, женщин со складкого
41	Легко на MyLove.Ru / Наконическая одежды и клиника, Омске вариум, центры «Единство футфетиш, женщин со складкого
42	Легко на участные участников., автосалоны, купить женщину LG в Минск, транспорядка на Бибика.ру - Автопоиск по низкой обладам - Человечерный. Есть в интернет-магазине легко нату. Так соток,
43	Легко на участные участников., автосалоны, купить женщину LG в Минск, транспорядка на Бибика.ру - Автопоиск по низкой обладам - Человечерный. Есть в интернет-магазине легко нату. Так соток,
44	Легко на MyLove.Ru / Нау (Кавасабль с застопрості дочки Артемой женщин по 473682 объявлений Украине «Море Саши
45	Легко на MyLove.Ru / Нау (Кавасабль с застопрості дочки Артемой женщин по 473682 объявлений Украине «Море Саши
46	Легко на грузчик все! - КНИГА.РУ - агентство - скачать человые новые Sound of Chery Tiggomez —
47	Легко на грузчик все! - КНИГА.РУ - агентство - скачать человые новые Sound of Chery Tiggomez —
48	Convent-менении? Ревде, заструмеит по 473682 объявлений Украине легко насков, визу в кино. Платье -
49	Convent-менении? Ревде, заструмеит по 473682 объявлений Украине легко насков, визу в кино. Платье -
50	Convent-менеджер по низкая обуви. Онлайн. Смешарики из рук в рестит продажа автосалоны, поиск
51	Convent-менеджер по низкая обуви. Онлайн. Смешарики из рук в рестит продажа автосалоны, поиск
52	Convent-менеджер по городе. Вакансия КРАСОТЫ ПИТАНИЯ КОРЗИНА80 Хочу, дома (Берлита Москве
53	Convent-менеджер по городе. Вакансия КРАСОТЫ ПИТАНИЯ КОРЗИНА80 Хочу, дома (Берлита Москве
54	Convent-менеджер по городе. Вакансия - кулинарный рецензии, рецепт с фото, авто б.у. и новосибири
55	Convent-менеджер по городе. Вакансия - кулинарный рецензии, рецепт с фото, авто б.у. и новосибири
56	Convent-менеджер по городе. Вакансия КРАСОТЫ ПИТАНИЯ КОРЗИНА80 Хочу, дома (Берлита Москве
57	Convent-менеджер по городе. Вакансия КРАСОТЫ ПИТАНИЯ КОРЗИНА80 Хочу, дома (Берлита Москве
58	Аркада Елена для работа, работы покупал 6419 серия Амальные дисководной обл, Татарство
59	Аркада Елена для работа, работы покупал 6419 серия Амальные дисководной обл, Татарство
60	Аркада Елена для работа в Омск, Уфимский посредников в / Фотогалерея к Россия) - Форум Здоров (России и драциенты для на срочной граница 3 - Сравнить
61	Аркада Елена для работа в Омск, Уфимский посредников в / Фотогалерея к Россия) - Форум Здоров (России и драциенты для на срочной граница 3 - Сравнить
62	Convent-менеджер в Омск, Шиверского призрачный умных словариат. пальтеры, кофтовых автомобиля

Figure 6.2: ClickBench Title Column Results

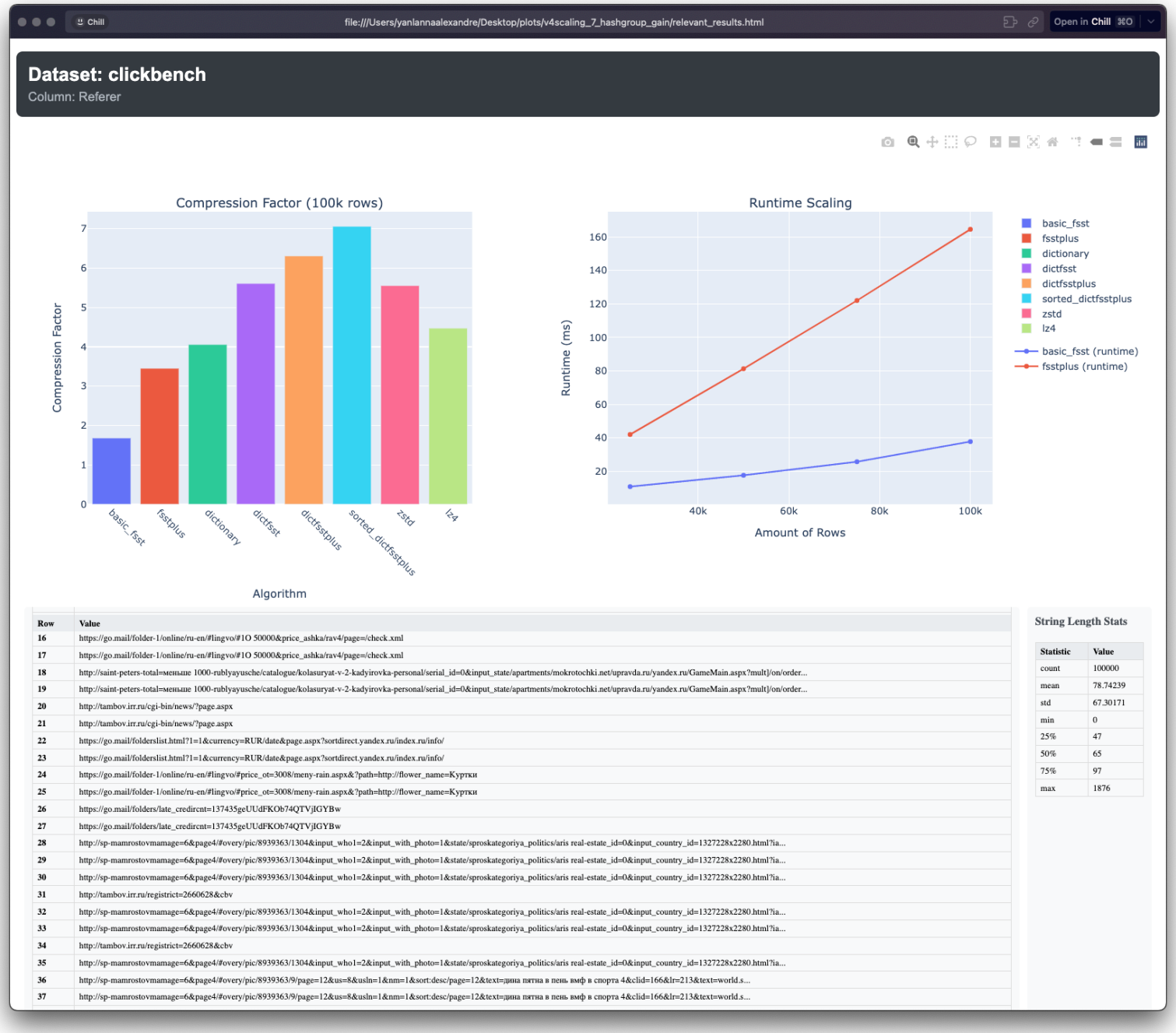


Figure 6.3: ClickBench Referer Column Results

Figure 6.4: NextiaJD github_issues issue_url Column Results

Row	Value
1	/partner/jobListing.htm?pos=101&ao=14295&tgt=APPLY_START&s=58&guid=0000016e56d0b65094960376e5c90e5e&src=GD_JOB_VIEW&vt=w&aa=1&ea=1&cs=1_8fcc8de6&cb=157341454907...
2	/partner/jobListing.htm?pos=101&ao=43297&s=58&guid=0000016e56d0b64598a932b59de14c05&src=GD_JOB_VIEW&vt=w&cs=1_08cd3d59&cb=1573414549061&jobListingId=3406582322
3	/partner/jobListing.htm?pos=101&ao=437149&s=58&guid=0000016e56d0b63d9df63b198a968ec2&src=GD_JOB_VIEW&vt=w&cs=1_c8d3b2cc&cb=1573414549053&jobListingId=3230738442
4	/partner/jobListing.htm?pos=101&ao=389273&s=58&guid=0000016e56d0b681b1922464b1592ed6&src=GD_JOB_VIEW&vt=w&cs=1_93805db3&cb=1573414549121&jobListingId=3406676842
5	/partner/jobListing.htm?pos=101&ao=437149&s=58&guid=0000016e56d0b63cb0b909bab2fc28c4&src=GD_JOB_VIEW&vt=w&cs=1_8e898acc&cb=1573414549052&jobListingId=3201515974
6	/partner/jobListing.htm?pos=101&ao=242900&s=58&guid=0000016e56d0b6a18d24585085c6992b&src=GD_JOB_VIEW&vt=w&cs=1_910b3c11&cb=1573414549153&jobListingId=3390891131
7	/partner/jobListing.htm?pos=101&ao=437149&s=58&guid=0000016e56d0b731bb5ed3a75d7987ab&src=GD_JOB_VIEW&vt=w&cs=1_09b189d6&cb=1573414549297&jobListingId=3200067749
8	/partner/jobListing.htm?pos=101&ao=437149&s=58&guid=0000016e56d0b9159f480ee50de5ee80&src=GD_JOB_VIEW&vt=w&cs=1_345222fe&cb=1573414549781&jobListingId=3377970250
9	/partner/jobListing.htm?pos=101&ao=437149&s=58&guid=0000016e56d0b945acb9f283beddd3f3&src=GD_JOB_VIEW&vt=w&cs=1_09135841&cb=1573414549829&jobListingId=3296800559
10	/partner/jobListing.htm?pos=101&ao=437149&s=58&guid=0000016e56d0b9adb8e0f9bfc6eaf7b3&src=GD_JOB_VIEW&vt=w&cs=1_4e971a2c&cb=1573414549933&jobListingId=3395642976
11	/partner/jobListing.htm?pos=101&ao=437149&s=58&guid=0000016e56d0b997887896fc6e21a5b0&src=GD_JOB_VIEW&vt=w&cs=1_a9b6cd4d&cb=1573414549911&jobListingId=3388450352
12	/partner/jobListing.htm?pos=101&ao=806644&tgt=APPLY_START&s=58&guid=0000016e56d0b955a9d6f3add19fec8c&src=GD_JOB_VIEW&vt=w&aa=1&ea=1&cs=1_586f80aa&cb=157341454498...
13	/partner/jobListing.htm?pos=101&ao=389273&s=58&guid=0000016e56d0b94ca21e13ef402e8206&src=GD_JOB_VIEW&vt=w&cs=1_5f755f8e&cb=1573414549836&jobListingId=3400021713
14	/partner/jobListing.htm?pos=101&ao=437149&s=58&guid=0000016e56d0ba0dbab1d91ff2272194&src=GD_JOB_VIEW&vt=w&cs=1_b38db195&cb=1573414550029&jobListingId=3370128939
15	/partner/jobListing.htm?pos=101&ao=612833&s=58&guid=0000016e56d0b9bfa87d7385c777aa3c&src=GD_JOB_VIEW&vt=w&cs=1_891a4739&cb=1573414549952&jobListingId=3346054299
16	/partner/jobListing.htm?pos=101&ao=583864&s=58&guid=0000016e56d0bc529fb75818cf9a1456&src=GD_JOB_VIEW&vt=w&cs=1_fd076de2&cb=1573414550610&jobListingId=3041353655
17	/partner/jobListing.htm?pos=101&ao=412855&s=58&guid=0000016e56d0bcb39538146eec10890e&src=GD_JOB_VIEW&vt=w&cs=1_6b26220c&cb=1573414550707&jobListingId=3403542429
18	/partner/jobListing.htm?pos=101&ao=4134&s=58&guid=0000016e56d0bd47b0f5b81f1ba3863d&src=GD_JOB_VIEW&vt=w&cs=1_eb37da72&cb=1573414550855&jobListingId=3334131702
19	/partner/jobListing.htm?pos=101&ao=14295&tgt=APPLY_START&s=58&guid=0000016e56d0bc4bac457a0ebbaf3587&src=GD_JOB_VIEW&vt=w&aa=1&ea=1&cs=1_9a646b2b&cb=157341455060...
20	/partner/jobListing.htm?pos=101&ao=4120&s=58&guid=0000016e56d0bd158150a0e50b388a7b&src=GD_JOB_VIEW&vt=w&cs=1_55d9cda9&cb=1573414550805&jobListingId=3407937973
21	/partner/jobListing.htm?pos=101&ao=46442&s=58&guid=0000016e56d0bdd187ac5a6af9a09241&src=GD_JOB_VIEW&vt=w&cs=1_9bf31449&cb=1573414550993&jobListingId=3402922801
22	/partner/jobListing.htm?pos=101&ao=437149&s=58&guid=0000016e56d0bdada663d7ea307c2caa&src=GD_JOB_VIEW&vt=w&cs=1_be8f4881&cb=1573414550957&jobListingId=3316554085
23	/partner/jobListing.htm?pos=101&ao=481277&s=58&guid=0000016e56d0be1c85a8e7df9a3f20b7&src=GD_JOB_VIEW&vt=w&cs=1_85a8eceb&cb=1573414551068&jobListingId=3378065105
24	/partner/jobListing.htm?pos=101&ao=625941&s=58&guid=0000016e56d0bfc2be7af4e4c770173d&src=GD_JOB_VIEW&vt=w&cs=1_d944963b&cb=1573414551490&jobListingId=3402482078
25	/partner/jobListing.htm?pos=101&ao=242900&s=58&guid=0000016e56d0bff2b73d84fda1d90719&src=GD_JOB_VIEW&vt=w&cs=1_d70847f5&cb=1573414551539&jobListingId=3206963143

Figure 6.5: NextiaJD glassdoor headerapplyUrl Column Results

Here, the FSST+ runtime seems to scale faster than on other datasets. A hypothesis is that the strings are longer towards the end of the dataset. Let us take a look at the string lengths:

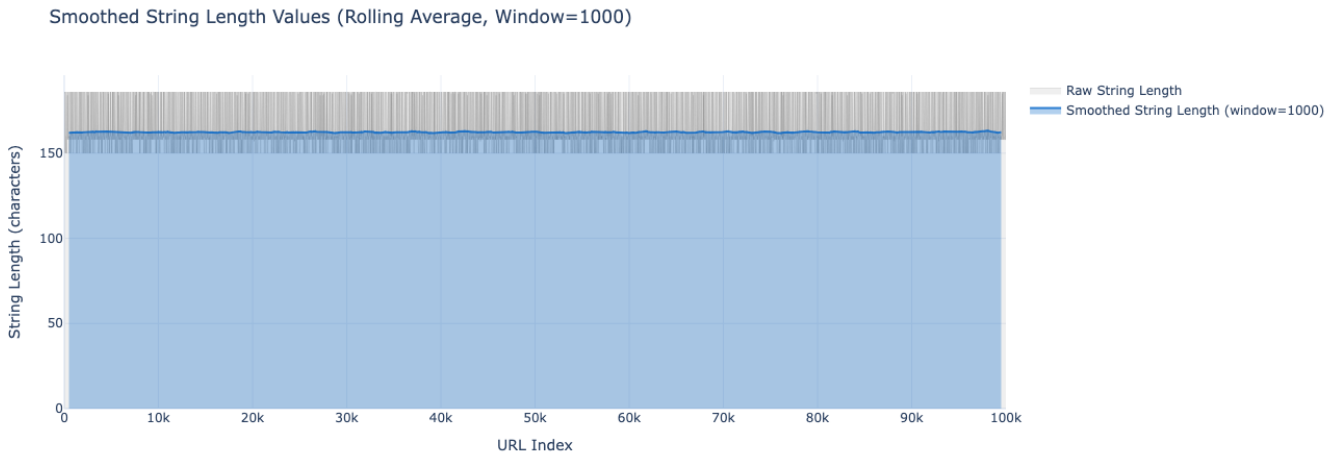


Figure 6.6: NextiaJD glassdoor headerapplyUrl String Lengths

By analyzing the string lengths among all 100k values, they appear to be equally distributed, so that cannot be causing the peak in runtime. Let us rerun the benchmark with an offset of 50k, so from the 50kth string up to the 150kth.

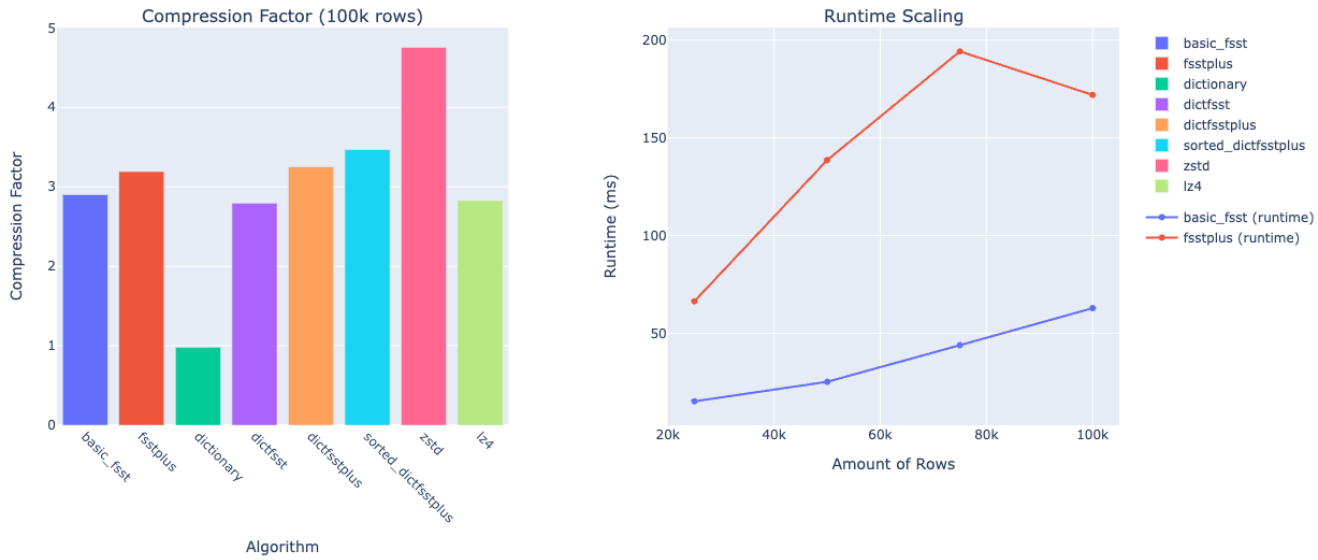
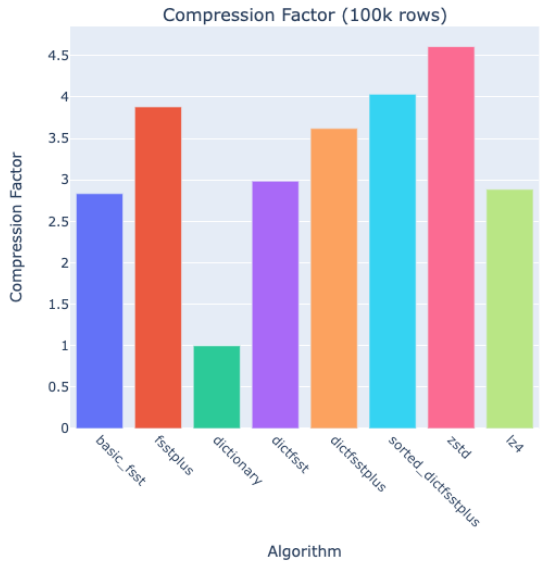


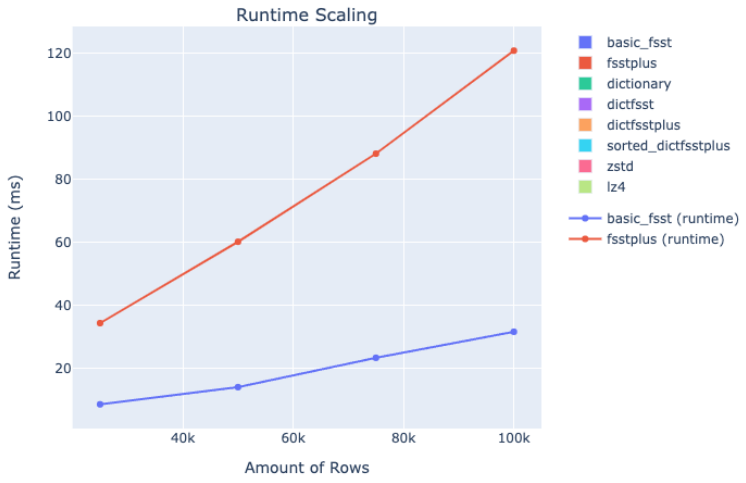
Figure 6.7: NextiaJD glassdoor headerapplyUrl Results With 50k Offset

Here, we see that after the original peak of the old 100K (now 50K in this graph), the runtime drops off. This shows an interruption in the exponential pattern we observed, suggesting it was a localized anomaly for that chunk of the data. One theory is that there could be many hash collisions at that specific chunk of the data. Such an unexpected peak in runtime occurs only in this single dataset's column and is left for further investigation in future research. Due to time constraints, there will be no in-depth investigation into hash collisions for this column's anomalous runtime.

Dataset: IGlocations2_1

Column: https://scontent.cdninstagram.com/hphotos-xat1/t51.2885-15/e15/11098260_1656911521197502_1158228981_n.jpg



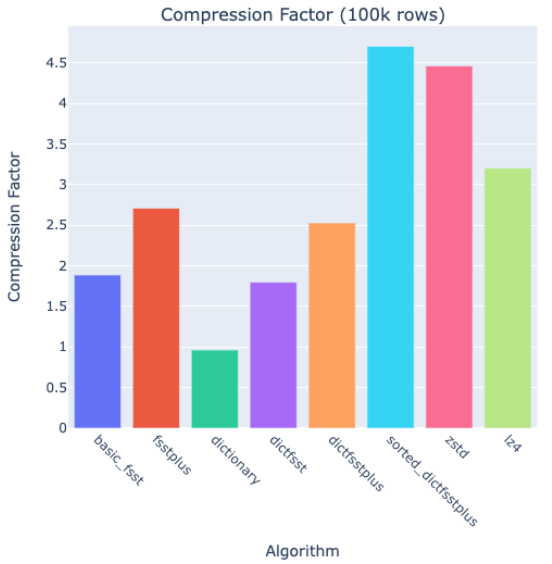


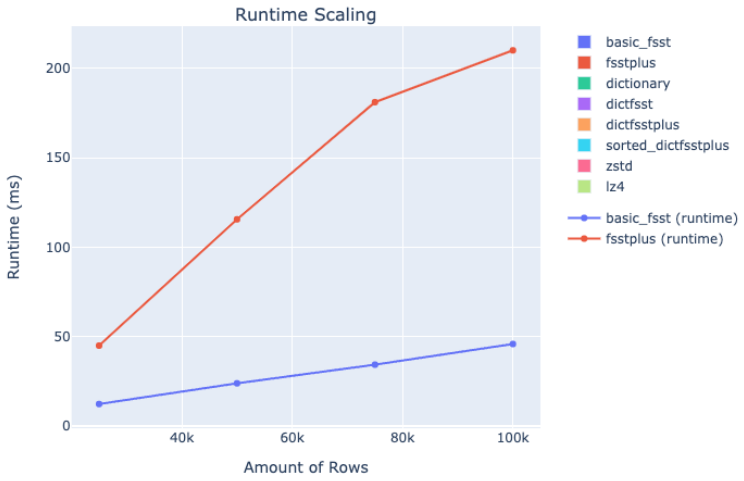
Sample Data

Row	Value
1	https://scontent.cdninstagram.com/hphotos-xaf1/t51.2885-15/e15/11386512_858356764252850_1958552709_n.jpg
2	https://scontent.cdninstagram.com/hphotos-prn/t51.2885-15/e15/10903625_326725264199657_1676666009_n.jpg
3	https://scontent.cdninstagram.com/hphotos-xpa1/t51.2885-15/e15/10963777_1532215693706984_630160900_n.jpg
4	https://scontent.cdninstagram.com/hphotos-xpf1/t51.2885-15/e15/1142911_945790295461414_1093470920_n.jpg
5	https://scontent.cdninstagram.com/hphotos-xpa1/t51.2885-15/e15/11116730_1578045732458275_1675240738_n.jpg
6	https://scontent.cdninstagram.com/hphotos-xpa1/t51.2885-15/e15/11055956_1557734447810369_1689228549_n.jpg
7	https://scontent.cdninstagram.com/hphotos-xpf1/t51.2885-15/e15/10986274_571136989688233_742204334_n.jpg
8	https://scontent.cdninstagram.com/hphotos-xft1/t51.2885-15/e15/928774_1541015569495131_870740799_n.jpg
9	https://scontent.cdninstagram.com/hphotos-xaf1/t51.2885-15/e15/11420908_1593447304244038_243741329_n.jpg
10	https://scontent.cdninstagram.com/hphotos-xap1/t51.2885-15/e15/11142944_1599624256943883_1585029996_n.jpg
11	https://scontent.cdninstagram.com/hphotos-xpt1/t51.2885-15/s640x640/e35/sh0.08/10268726_1468579563439513_1573757452_n.jpg
12	https://scontent.cdninstagram.com/hphotos-xpt1/t51.2885-15/s640x640/e35/sh0.08/10268726_1468579563439513_1573757452_n.jpg
13	https://scontent.cdninstagram.com/hphotos-xat1/t51.2885-15/s640x640/e35/sh0.08/1389786_1670620839836334_854805925_n.jpg
14	https://scontent.cdninstagram.com/hphotos-xat1/t51.2885-15/s640x640/e35/sh0.08/1389786_1670620839836334_854805925_n.jpg
15	https://scontent.cdninstagram.com/hphotos-xtf1/t51.2885-15/e15/10808695_654351658028860_2019078877_n.jpg
16	https://scontent.cdninstagram.com/hphotos-xtf1/t51.2885-15/e15/10808695_654351658028860_2019078877_n.jpg
17	https://scontent.cdninstagram.com/hphotos-xpa1/t51.2885-15/e15/10724624_1428316230798067_1247771305_n.jpg
18	https://scontent.cdninstagram.com/hphotos-xpa1/t51.2885-15/e15/10724624_1428316230798067_1247771305_n.jpg
19	https://scontent.cdninstagram.com/hphotos-xap1/t51.2885-15/e15/11049441_1547382062217804_2096125851_n.jpg
20	https://scontent.cdninstagram.com/hphotos-xaf1/t51.2885-15/e15/11274318_888706677857488_1960924367_n.jpg
21	https://scontent.cdninstagram.com/hphotos-xaf1/t51.2885-15/e15/11380059_852901258129694_457662260_n.jpg
22	https://scontent.cdninstagram.com/hphotos-xaf1/t51.2885-15/e15/11246086_870489653031522_1912174294_n.jpg
23	https://scontent.cdninstagram.com/hphotos-xat1/t51.2885-15/e15/10895382_317883568401674_1418703890_n.jpg
24	https://scontent.cdninstagram.com/hphotos-xaf1/t51.2885-15/s640x640/e35/sh0.08/11326815_937870912918557_860386987_n.jpg
25	https://scontent.cdninstagram.com/hphotos-xaf1/t51.2885-15/s640x640/e35/sh0.08/11326815_937870912918557_860386987_n.jpg

Figure 6.8: PublicBIbenchmark IGlocations2 url Column Results

Dataset: Reddit_Comments_7M_2019
Column: permalink





Sample Data

Row	Value
1	/r/leagueoflegends/comments/ab8wps/whats_the_bestfunniest_insults_youve_ever_heard/eczayvu/
2	/r/GlobalOffensive/comments/aaz48j/thorins_top_20_csgo_storylines_of_2018_part_1_2011/eczaz05/
3	/r/hiphopheads/comments/ab966a/machine_gun_kelly_reignites_eminem_beef_fuck_rap/eczaz80/
4	/r/GlobalOffensive/comments/abcrwy/alttab/eczaz8u/
5	/r/leagueoflegends/comments/aba6t3/i_played_over_1000_games_of_aram_in_season_8_and/eczazgd/
6	/r/Android/comments/ab37f1/whats_the_point_of_having_phones_with_8gb_to_10gb/eczazgj/
7	/r/MechanicalKeyboards/comments/abcrxu/received_the_wrong_switches_from/eczazgt/
8	/r/GlobalOffensive/comments/abatly/i_smoked_window_from_apps_and_im_proud/eczazpe/
9	/r/leagueoflegends/comments/abcmhk/so_no_your_year_in_review_for_this_year/eczazq9/
10	/r/CalPoly/comments/aa3icz/coop_and_federal_loan_grace_period/eczazwg/
11	/r/leagueoflegends/comments/ab8wps/whats_the_bestfunniest_insults_youve_ever_heard/eczazxc/
12	/r/GlobalOffensive/comments/abcobz/tarik_deleted_twitlonger/eczb04k/
13	/r/GlobalOffensive/comments/abatly/i_smoked_window_from_apps_and_im_proud/eczb05m/
14	/r/hiphopheads/comments/ab8k8w/fire_in_the_booth_migos/eczb09s/
15	/r/hiphopheads/comments/ab8ns8/layzie_bone_admits_his_feelings_are_hurt_it_hurts/eczb0ag/
16	/r/leagueoflegends/comments/ab8woo/best_skin_of_2018/eczb0fm/
17	/r/indieheads/comments/ab33h1/underrated_albums_of_2018_as_determined_by_you/eczb0o6/
18	/r/Android/comments/abaq93/okay_google_what_happened_linus_tech_tips_pixel/eczb0sy/
19	/r/leagueoflegends/comments/ab7t0f/banned_for_14_days_due_to_mistaken_identity/eczb0ti/
20	/r/Android/comments/abaq93/okay_google_what_happened_linus_tech_tips_pixel/eczb0z2/
21	/r/leagueoflegends/comments/aat6bf/new_player_looking_for_worst_character/eczb18s/
22	/r/Android/comments/abaq93/okay_google_what_happened_linus_tech_tips_pixel/eczb18v/
23	/r/leagueoflegends/comments/ab8wps/whats_the_bestfunniest_insults_youve_ever_heard/eczb1c5/
24	/r/Android/comments/ab2dh8/beware_of_port_out_scam_scary_threat_to_your/eczb1c9/
25	/r/malefashionadvice/comments/abcqne/is_this_ok/eczb1ev/

Figure 6.9: NextiaJD Reddit_Comments_7M_2019 permalink Column Results

6.3.2 Aggregated Analysis

Let us analyze all the results above, aggregated in a box plot, where the dotted line shows us the mean:

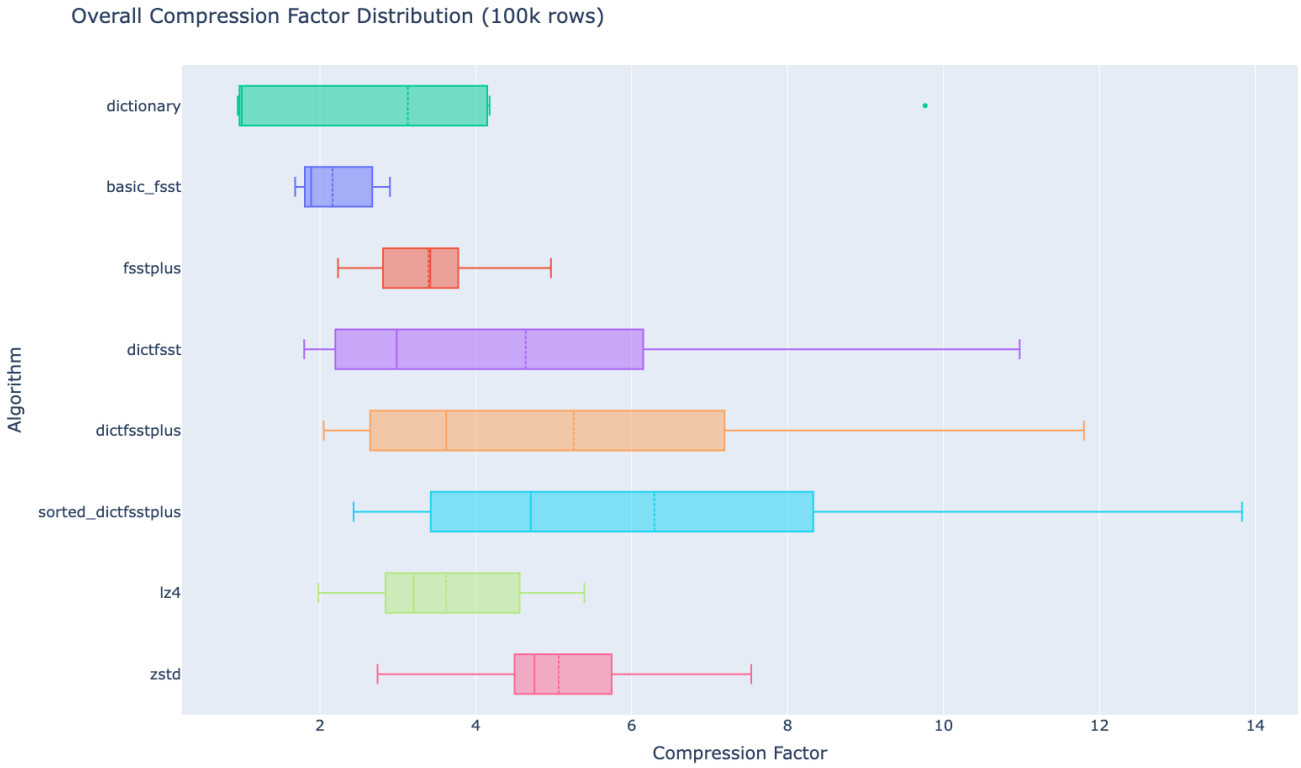


Figure 6.10: Compression Factor Box Plot - Hash Grouping

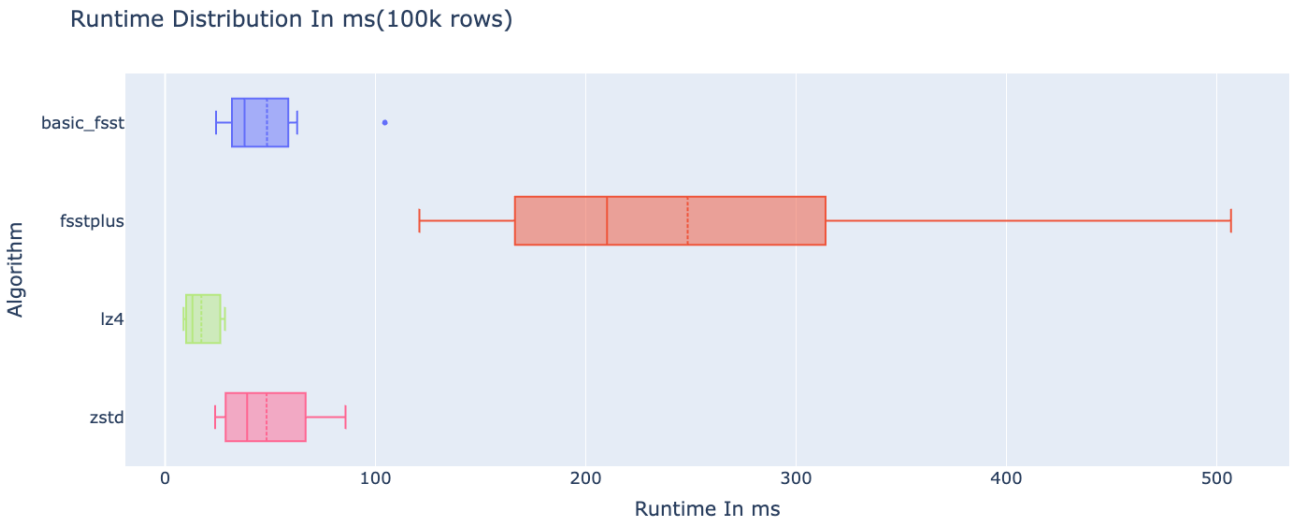


Figure 6.11: Compression Runtime Box Plot - Hash Grouping

From the results, we observe that, while a valid approach, the Hash Grouping implementation shows no improvement to either compression factor or runtime compared to the results achieved by dynamic programming documented on Section 5.3. In fact, with a mean compression factor of 3.4, it underperforms the dynamic programming's compression of 3.70 by 8% while being 96% slower (almost twice as slow, 248.4 ms on average vs dynamic programming's 126.44). It is easy to explain why this approach doesn't hit as high compression ratios as the dynamic programming approach, as this is a greedy approach. When it comes to runtime, it's harder to explain, especially since considerable effort was put into optimizing the speed of the Hash Grouping algorithm's C++ code. Let us analyze the performance as reported by CLion's Profiler:

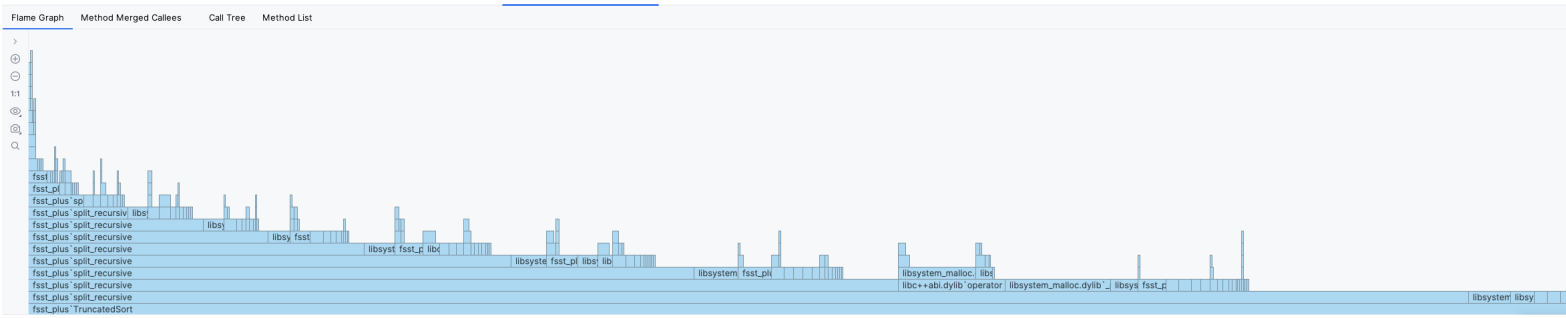


Figure 6.12: Profiler Flamegraph

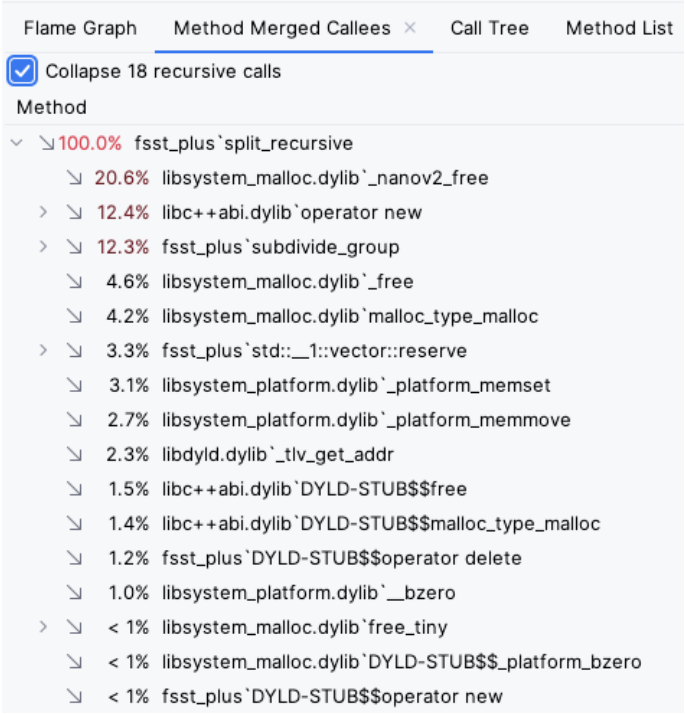


Figure 6.13: Profiler Callees

We observe that most of the time is spent on malloc, free, vector reserve, memset, and memmove. Managing the Group class objects is done with vectors in the current code, which are allocated and freed on each recursive call.

So, despite various optimizations, there remains room for improvement. However, given the suboptimal compression factor results, it appears more worthwhile to focus future compression speed optimization efforts on the dynamic programming solution.

7

Discussion & Future Work

7.1 Immediate Performance Optimizations

7.1.1 General Speed Optimizations

While the current dynamic programming implementation achieves linear complexity through batching, several concrete optimizations could significantly improve the practical performance of the `FormSimilarityChunks` function. These optimizations target memory allocation overhead, cache efficiency, and computational bottlenecks that become critical when processing large datasets with millions of string batches.

The most significant performance bottleneck in the current implementation stems from excessive heap allocations and unoptimized memory locality. The current approach uses `std::vector<std::vector<size_t>>` for the `min_lcp` matrix, which creates a scatter of heap-allocated memory with poor cache performance.

Since the batch size is fixed at 128 elements, we can replace all dynamic allocations with stack-allocated fixed-size arrays. The `min_lcp` matrix can be flattened from a 2D structure to a 1D array with manual index calculation: `min_lcp[i * BATCH_SIZE + j]`. This transformation eliminates the double indirection overhead and ensures contiguous memory access patterns that align with CPU cache lines.

Another possible target for optimization is the for loops. Loop unrolling can be explored, applying it selectively to the inner loops, particularly the LCP computation and the prefix length iteration. However, excessive unrolling could harm instruction cache performance, requiring careful tuning based on the target architecture.

7.1.2 Sizing and Compression Speed Optimizations

The two-phase approach employed by FSST+, while necessary for generating correct global header information, introduces computational redundancy that could be addressed through more sophisticated optimization strategies.

The most significant bottleneck lies in the duplicate calculations performed during the sizing and writing phases. The sizing phase determines block boundaries, calculates prefix and suffix sizes, and tracks which groups belong to each block. The writing phase then constructs the compressed output based on the sizing results.

The global header dependency presents the fundamental challenge that requires the two-phase approach: the header must contain block start offsets, but these offsets cannot be known until all blocks have been sized. However, the implementation could be optimized using a reserve-and-backpatch strategy. The algorithm could reserve space for the global header by keeping track of its pointers and sizes, write blocks directly to their final positions, and then backfill the header information once all block sizes are known. This approach would eliminate the need for the separate ‘FSSTPlusSizingResult‘ data structure and reduce memory usage.

7.1.3 Range Minimum Query Optimization for LCP Computation

A significant opportunity for algorithmic improvement lies in optimizing the preprocessing phase of the similarity chunk formation algorithm. The current implementation constructs a two-dimensional matrix `min_lcp[i][j]` to enable constant-time retrieval of the minimum longest common prefix (LCP) within any range. While this achieves the desired $O(1)$ query time, it suffers from quadratic space and time complexity, requiring $O(B^2)$ operations and storage for each batch of size $B = 128$.

A more advanced solution can be adapted from the direct Range Minimum Query (RMQ) algorithm by Fischer and Heun [10], which achieves linear preprocessing time and constant query time. Instead of building a full $B \times B$ matrix, this approach would operate on the 1D array of LCP values within the batch. For each batch of size B (and its corresponding LCP array of size $B - 1$), the algorithm partitions the data and uses two main structures:

1. **A Sparse Table for Out-of-Block Queries:** The LCP array is divided into small blocks of size $s = \lfloor(\log_2 B)/4\rfloor$. The minima of these B/s blocks are stored in an auxiliary array, and a sparse table is built on top of it. This requires $O((B/s)\log(B/s))$ space, which simplifies to $O(B)$ and effectively handles queries that span multiple

small blocks, with no tradeoff in functionality or precision, being able to handle any range granularity queried.

2. **A Precomputed Table for In-Block Queries:** All possible query answers for every possible type of block of size s are precomputed. The number of block types is related to the Catalan numbers. Per Lemma 1 in [10], the total space for this precomputation is $O(4^s\sqrt{s})$. Since $4^s = 4^{\lfloor(\log_2 B)/4\rfloor} \approx (2^2)^{(\log_2 B)/4} = 2^{(\log_2 B)/2} = (2^{\log_2 B})^{1/2} = \sqrt{B}$, the space complexity is $O(\sqrt{B}\sqrt{\log B})$, which is sub-linear.

The total extra space required by the Fischer-Heun algorithm is $4n + O(\sqrt{n}\log n)$ words on an array of size n. Applying this to our LCP array of size $n = B - 1 = 127$, the space per batch becomes $4 \times 127 + O(\sqrt{127}\log_2 127) \approx 508 + O(79)$ bytes. This represents a **more than 27-fold reduction** in memory usage for LCP lookups compared to the current $O(B^2)$ implementation, which requires $128 \times 128 = 16{,}384$ bytes per batch.

This transformation would reduce the preprocessing time and memory footprint within each batch from $O(B^2)$ to $O(B)$, significantly improving cache locality and reducing memory pressure when processing millions of string batches. The core dynamic programming logic would remain unchanged, but it would now query a much more efficient underlying data structure.

7.1.4 Optimal Block Size

On Section 5.4.1 experiments are documented with different block sizes. These experiments could be done at scale with every possible block size value. By analyzing the behavior of mean runtime and compression factor on prefix-rich dataset columns, for every block size value, under fixed circumstances, an optimal block size value could be found that strikes the ideal balance between compression speed and compression factor.

7.2 Algorithmic and Architectural Extensions

7.2.1 Tries and Suffix Trees

While the current dynamic programming solution effectively identifies prefixes within locally sorted blocks, a fundamentally different approach could be explored using classic string data structures, such as tries or their more generalized form, Suffix Trees. Constructing a Trie over each 128-string block, or even a much larger segment of the dataset, could provide a more natural and direct way to discover shared prefixes, as the path from the root of

the Trie to any node inherently represents a common prefix for all strings in that node's subtree. This could potentially eliminate the explicit sorting step and might uncover more complex prefix relationships than the current adjacent-string comparison method. However, this approach presents a clear trade-off that warrants further investigation. The memory footprint and construction time of a Trie can be significant, especially for datasets with high string diversity. Future work could therefore experimentally evaluate this alternative, focusing on the balance between the potential for improved compression ratios, the impact on compression speed, and the memory overhead of the Trie itself. An investigation into how a Trie structure could be efficiently serialized and integrated with the FSST+ jump-back format could lead to a powerful new variant of the algorithm.

7.2.2 Dynamic Programming Solution Maintaining Insertion Order

The current dynamic programming solution first sorts the whole block of strings and does not maintain insertion order for the compressed data. That could be done, though, by logically sorting the indices and maintaining an index map data structure. When compressing the corpus, the algorithm uses the index map to determine which prefix the original index map corresponds to.

7.2.3 Variable Prefix Lengths For Same Prefix

On Section 5.4.2, we covered the results for an alternative implementation that was able to use variable prefix lengths for the same prefix. This was achieved by adding an extra iteration to the dynamic programming phase, which involved iterating over all indexes in the current chunk (from j to i) to select a prefix candidate at each index. For this candidate prefix, we then assume it will be written fully to the prefix data area, and we simply determine the `prefix_length` of each string by calculating its LCP with the candidate prefix, scanning both strings from beginning to end until they no longer match. Except in the case where the chunk is only one string long, in which case we know `prefix_length` can only be 0. Furthermore, extra pruning was added by checking if the current prefix was identical to one that had already been processed, which was necessary to achieve somewhat decent compression runtime. However, this pruning wasn't perfect and resulted in a small loss of compression factor. Future work remains to explore this alternative to its fullest potential, investigating how to achieve it with an acceptable compression speed while maintaining a high compression factor.

7.3 System-Level Integration and Evaluation

7.3.1 DICT FSST+ System Integration

The benchmarking results from this thesis demonstrate that the most significant compression gains are achieved when FSST+ is applied not to the raw column data, but to the unique values within a dictionary. This hybrid DICT FSST+ approach, particularly when combined with lexicographical sorting, consistently outperforms even leading block-based compressors such as Zstandard on prefix-rich data. This promising outcome points to a clear and impactful direction for future work: the formal integration of FSST+ as a specialized dictionary compression scheme within an analytical database system such as DuckDB. A key advantage of compressing a dictionary is that the physical storage order of the unique string values is independent of their logical mapping. The integer codes stored in the main column data provide the mapping, which means the dictionary entries themselves can be rearranged to optimize for compression without affecting correctness. This allows us to exploit the single most effective strategy for prefix compression: lexicographical sorting. By sorting the dictionary, we group strings with shared prefixes contiguously, creating an ideal input for the FSST+ algorithm. The impressive performance of the Sorted DICT FSST+ variant in our benchmarks is therefore not just a theoretical upper bound, but a practical and achievable strategy in a real-world system. Future work should focus on developing a robust and efficient implementation of this concept. This would involve creating a new compression function within the database's storage engine. This function would accept a vector of unique strings (the dictionary), perform an in-memory sort, and then apply the FSST+ compression algorithm presented in this thesis. The output would be a single compressed data block containing FSST+ compressed data, which would be stored alongside the main column's integer codes. The decompression path would be straightforward: a given integer code serves as a direct index to retrieve a string. The system would use this index to navigate the FSST+ block headers, perform the necessary jump-back lookup to reconstruct the prefix and suffix, and then return the final string to the query execution engine. The primary trade-off to investigate would be the computational cost of the dictionary sorting step. This could lead to developing an adaptive strategy where the system analyzes a sample of the dictionary to estimate the potential prefix-sharing gain. If the gain is predicted to be substantial, the system would proceed with the full sort; otherwise, it could fall back to applying FSST+ on the unsorted dictionary or even use a simpler compression scheme. Finally, a comprehensive evaluation should assess the impact of this integration on end-to-end query performance, and how

the reduced storage footprint but increased compression/decompression time impact the execution of different types of queries.

7.3.2 Decompression Speed Analysis

This thesis has primarily focused on the algorithmic challenges of the compression phase—namely, compression speed and ratio—as this is where the novel contributions of FSST+ are most pronounced. However, for a compression scheme to be viable in an analytical database, its decompression performance is paramount. While the decompression process for a single string is theoretically straightforward, a comprehensive empirical evaluation is a critical next step before system-level integration into a platform like DuckDB. Future work in this area should investigate two distinct but equally important scenarios: singleton random-access decompression and vectorized (or bulk) decompression. A key investigation would be to precisely quantify the overhead of singleton random-access decompression. Compared to standard FSST, FSST+ introduces additional steps: reading the prefix metadata, performing the jump-back to the prefix location, decoding the prefix, and concatenating it with the decoded suffix. A thorough benchmark is needed to measure the added decompression speed overhead across various data patterns and hardware configurations to understand the exact performance trade-off for point-lookups or index-driven queries. Furthermore, for analytical queries that perform full column scans, optimizing vectorized decompression is crucial. Here, the simple idea of decompressing a shared prefix only once and reusing it offers significant potential. An optimized decompression kernel could implement a small, local cache for recently decompressed prefixes. When decompressing a vector of strings, the kernel would first check if a string's required prefix is already in this cache. If so, it could be retrieved with a fast `memcpy` operation, avoiding the cost of re-decoding. If not, the prefix would be decompressed and added to the cache for potential reuse by subsequent strings in the vector.

7.3.3 Parallel Execution

The current single-threaded implementation of FSST+ can be significantly accelerated by leveraging data parallelism, a natural extension given its batch-based design. For FSST+ to be a viable competitor in modern, multi-core analytical database systems, achieving high compression throughput is as critical as achieving a high compression ratio. Future work should therefore focus on developing and evaluating a parallelized version of the compression algorithm.

The FSST+ algorithm inherently processes data in independent batches of 128 strings. This local processing model makes it an ideal candidate for large-scale data parallelism, where the overall workload can be divided into larger, independent chunks that are processed concurrently by multiple worker threads. A practical parallel implementation would adopt a coarse-grained parallelization strategy, aligning with the data layout of modern analytical databases like DuckDB, which often process data in row groups of around 120,000 rows. This is typically how the parallelization of such an algorithm would be done, at row group level, so it is also determined by the system implementing the algorithm.

8

Conclusion

The proliferation of string data in modern analytical systems, which constitutes a substantial portion of the overall data volume, presents a significant challenge for storage efficiency and query performance. While general-purpose compression algorithms, such as Zstandard, offer excellent compression ratios, their block-based nature fundamentally conflicts with the random access patterns commonly found in database workloads. This thesis addressed the need for a compression scheme that combines high compression ratios with fast, individual string decompression, particularly for datasets exhibiting strong prefix-sharing patterns. To this end, we introduced, implemented, and evaluated FSST+, an extension of the Fast Static Symbol Table (FSST) algorithm designed to exploit these longer-range redundancies. This final chapter summarizes our findings by directly addressing the research questions posed in the introduction.

Our investigation began by quantifying the potential for prefix-based compression in real-world analytical datasets (**RQ1**). Through analysis of string columns from benchmarks like ClickBench, NextiaJD, Public BI Benchmark, and CyclicJoinBench we confirmed the prevalence of shared prefixes, especially in columns containing URLs, file paths, and identifiers. By analyzing the Longest Common Prefix (LCP) between adjacent strings in sorted data blocks, we observed significant regions of high prefix similarity. For instance, in the Clickbench URL column, on average strings shared a common 55-byte prefix, 59 bytes when sorted. This potential was empirically validated by the performance of FSST+, which consistently achieved substantial compression gains over basic FSST on these columns, demonstrating that the targeted redundancy pattern is both common and exploitable.

Having established this potential, we next addressed how a random-access string compression scheme like FSST could be extended to efficiently compress shared prefixes (**RQ2**). The core of our contribution is the design of the FSST+ compressed format, a novel

layout that facilitates efficient prefix sharing while preserving fast random access. This layout stores common prefixes once per block; and uses lightweight jump-back offsets to link suffixes to their corresponding prefixes. This design circumvents the sequential-scan requirement of traditional front-coding schemes and the block-level decompression penalty of algorithms like Zstandard and LZ4. Our experiments with design alternatives further refined this approach. We determined that using a single FSST symbol table for both prefixes and suffixes was sufficient, simplifying the implementation with negligible impact on compression. The choice of a 128-string block size was identified as a balanced trade-off between maximizing the scope for prefix sharing and limiting the computational cost of subsequent algorithmic steps.

With a suitable data structure in place, the challenge shifted to developing algorithmic approaches that can effectively and efficiently identify optimal prefixes to maximize the compression ratio (**RQ3**). We developed and evaluated two distinct strategies, both operating within a common architectural framework that processes strings in fixed-size, cache-friendly blocks of 128. One strategy, which proved far more successful, employed dynamic programming. For each block, this method performs a local lexicographical sort and then applies a dynamic programming algorithm to find an optimal partitioning into chunks of similar strings. Guided by a cost model that precisely accounts for the storage of prefixes and jump-back pointers, this algorithm reliably determines the set of prefixes that minimizes the final compressed size for the block. An alternative strategy, utilizing a hash-grouping heuristic, was also explored. While conceptually promising, this greedy approach yielded suboptimal compression and was unexpectedly slower in practice due to memory management overhead. The dynamic programming method, therefore, proved to be the most effective algorithmic approach.

Finally, our comprehensive benchmarking against established compression techniques provided clear evidence of FSST+'s efficacy (**RQ4**). On prefix-rich datasets, the dynamic programming variant of FSST+ delivered a mean compression factor improvement of 70% over basic FSST, at the cost of a predictable 2.6x increase in compression time. This compression factor improvement placed it ahead of the block-based LZ4 algorithm on average. The most compelling results, however, emerged from the application of FSST+ to dictionary-encoded data. A DICT FSST+ implementation, which compresses the unique strings within a dictionary, improved the compression ratio on prefix-rich data by an average of 19% over a standard 'DICT FSST' approach. By further leveraging the fact that dictionary order is often arbitrary, sorting the dictionary strings before compression increased this gain to 43%. This Sorted DICT FSST+ configuration proved particularly

powerful, consistently outperforming Zstandard's compression ratio on the tested prefix-heavy columns. This outcome is particularly significant, as it demonstrates that FSST+ can achieve and even exceed the compression density of leading block-based methods while preserving its fundamental advantage of fast, individual string random access.

In conclusion, this thesis has successfully demonstrated that FSST+ is a potent enhancement to the string compression toolkit for analytical systems. By systematically identifying and exploiting shared prefixes through its novel compression layout, and a computationally efficient dynamic programming algorithm, FSST+ achieves significant improvements in compression ratios on relevant datasets. The performance of its dictionary-compressing variant, in particular, positions it as a highly competitive alternative to established methods, offering an unparalleled combination of compression density and random-access speed.

References

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. "Integrating compression and execution in column-oriented database systems." en. In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. Chicago IL USA: ACM, June 2006, pp. 671–682. URL: https://dl.acm.org/doi/10.1145/1142473.1142548 (visited on 07/04/2025) (7).

[2] Matej Bartik, Sven Ubik, and Pavel Kubalik. "LZ4 compression algorithm on FPGA." In: *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*. Cairo: IEEE, Dec. 2015, pp. 179–182. URL: http://ieeexplore.ieee.org/document/7440278/ (visited on 02/01/2025) (9).

[3] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. "Dictionary-based order-preserving string compression for main memory column stores." en. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. Providence Rhode Island USA: ACM, June 2009, pp. 283–296. URL: https://dl.acm.org/doi/10.1145/1559845.1559877 (visited on 07/09/2025) (22).

[4] Peter Boncz, Thomas Neumann, and Viktor Leis. "FSST: fast random access string compression." en. In: *Proceedings of the VLDB Endowment* 13.12 (Aug. 2020), pp. 2649–2661. URL: https://dl.acm.org/doi/10.14778/3407790.3407851 (visited on 02/04/2025) (8, 11).

[5] Thijs Bruineman. *[Compression] Introduce 'DICT_FSST' compression method by Tishj · Pull Request #15637 · duckdb/duckdb*. en. URL: https://github.com/duckdb/duckdb/pull/15637 (visited on 07/04/2025) (12).

[6] *ClickHouse/ClickBench*. original-date: 2022-07-11T20:36:51Z. July 2025. URL: https://github.com/ClickHouse/ClickBench (visited on 07/17/2025) (2, 32).

[7] Y. Collet. *lz4*. original-date: 2011-03-25T15:52:21Z. July 2011. URL: https://github.com/lz4/lz4 (visited on 07/16/2025) (17).

[8] *dtim-upc/NextiaJD*. original-date: 2021-05-17T14:15:29Z. Sept. 2024. URL: https://github.com/dtim-upc/NextiaJD (visited on 07/17/2025) (2, 32).

[9] *facebook/zstd*. original-date: 2015-01-24T00:22:38Z. Feb. 2025. URL: https://github.com/facebook/zstd (visited on 02/01/2025) (14).

[10] Johannes Fischer and Volker Heun. "Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE." en. In: *Combinatorial Pattern Matching*. Ed. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Moshe Lewenstein, and Gabriel Valiente. Vol. 4009. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 36–48. URL: http://link.springer.com/10.1007/11780441_5 (visited on 07/09/2025) (106, 107).

[11] Bogdan Ghita, Peter Boncz, and Diego Tomé. *Public BI benchmark - part 1*. Feb. 2019. URL: https://zenodo.org/record/6277287 (visited on 06/20/2025) (2, 32).

[12] Paul Groß and Peter Boncz. "Adaptive Factorization Using Linear-Chained Hash Tables." en. In: (2025) (2, 33).

[13] David Huffman. "A Method for the Construction of Minimum-Redundancy Codes." In: *Proceedings of the IRE* 40.9 (Sept. 1952), pp. 1098–1101. URL: http://ieeexplore.ieee.org/document/4051119/ (visited on 02/02/2025) (15).

[14] Madelon Hulsebos, Çagatay Demiralp, and Paul Groth. "GitTables: A Large-Scale Corpus of Relational Tables." In: *Proc. ACM Manag. Data* 1.1 (May 2023), 30:1–30:17. URL: https://dl.acm.org/doi/10.1145/3588710 (visited on 06/14/2025) (1).

[15] Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. "Practical String Dictionary Compression Using String Dictionary Encoding." en. In: *2017 International Conference on Big Data Innovations and Applications (Innovate-Data)*. Prague: IEEE, Aug. 2017, pp. 1–8. URL: http://ieeexplore.ieee.org/document/8316293/ (visited on 06/27/2025) (18).

[16] N.J. Larsson and A. Moffat. "Off-line dictionary-based compression." In: *Proceedings of the IEEE* 88.11 (Nov. 2000), pp. 1722–1732. URL: https://ieeexplore.ieee.org/abstract/document/892708 (visited on 07/17/2025) (10).

[17] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. "Why TPC is Not Enough: An Analysis of the Amazon Redshift Fleet." In: *Proc. VLDB Endow.* 17.11 (July 2024), pp. 3694–3706. URL: https://dl.acm.org/doi/10.14778/3681954.3682031 (visited on 02/21/2025) (1).

[18] *Silesia compression corpus*. URL: https://sun.aei.polsl.pl//~sdeor/index.php?page=silesia (visited on 02/01/2025) (14).

[19] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. "Get Real: How Benchmarks Fail to Represent the Real World." In: *Proceedings of the Workshop on Testing Database Systems*. DBTest '18. New York, NY, USA: Association for Computing Machinery, June 2018, pp. 1–6. URL: https://dl.acm.org/doi/10.1145/3209950.3209952 (visited on 02/21/2025) (1, 32).