

CAST: Columnar Agnostic Structural Transformation

A Schema-less Structural Preprocessing Algorithm for Improving General-Purpose
Compression on Structured Data

Andrea Olivari

January 2026

Abstract

General-purpose compression algorithms such as LZ77, LZMA, and Zstandard rely on finite dictionary windows and local pattern matching to detect redundancy. While effective for unstructured text, these approaches often fail to fully exploit the long-range structural regularity present in machine-generated data, including, but not limited to, CSV files, JSON documents, and system logs.

We introduce **CAST** (Columnar Agnostic Structural Transformation), a schema-less structural pre-processing algorithm that infers repetitive layouts directly from the input stream. CAST decomposes each record into a static structural template (*Skeleton*) and a sequence of dynamic values (*Variables*), reorganizing the latter into column-aligned streams prior to compression. This transformation reduces structural entropy and exposes redundancy that is poorly captured by standard compressors operating on row-oriented text.

Experimental evaluation across diverse real-world datasets shows that CAST consistently improves compression density and, in many cases, reduces end-to-end compression time when paired with existing back-end compressors such as LZMA2. On highly repetitive structured datasets, compression ratios significantly exceed those obtained by standalone compressors, demonstrating that lightweight, schema-free structural normalization can substantially enhance general-purpose compression without modifying the underlying encoding algorithms.

1 Introduction

Machine-generated data (logs, IoT telemetry, database dumps) is characterized by rigid and repetitive structural patterns. Columnar storage formats such as Apache Parquet and ORC exploit this property effectively, but require explicit schemas and are therefore impractical for ad-hoc, semi-structured, or heterogeneous text archives. Conversely, general-purpose stream compressors such as `xz` (LZMA2) and `zstd` operate without schemas but are constrained by finite dictionary windows and local redundancy detection, often missing long-range structural repetition.

CAST proposes a middle ground: a schema-less structural transformation that reorganizes row-oriented text into column-oriented streams inferred dynamically from the data itself. Rather than introducing a new compression algorithm, CAST operates as a pre-processing stage that reshapes the input to better align with the strengths of existing compressors.

This paper details the CAST algorithm: while the transformation is fundamentally backend-agnostic, this study evaluates its effectiveness specifically in combination with **LZMA2**, utilizing a performance-oriented Rust implementation (featuring a custom zero-copy parsing strategy).

We test this engine in two configurations: **native library integration** to measure strict algorithmic efficiency (Table 1), and **system-level backend (7-Zip)** to demonstrate real-world throughput. Notably, **this second configuration yields the best overall results**, achieving substantial speedups with negligible compression loss compared to the native integration, as demonstrated in Table 2.

A Python reference implementation is also provided to illustrate the core logic. While functionally analogous to the Rust version, it prioritizes readability via standard Regular Expressions and operates as a single-threaded process in its native configuration, making it distinct from the high-performance Rust engine.

2 Methodology

The fundamental premise of CAST is that structured text lines L can be decomposed into a static template S and a variable vector V :

$$L \rightarrow S + V$$

Standard compressors process S and V interleaved. CAST processes unique S sets (Skeletons) once, and V sets (Variables) as contiguous blocks grouped by column index.

2.1 Adaptive Pattern Recognition

The algorithm does not enforce a schema. Instead, it utilizes a **specialized parsing engine** to process lines. To accommodate different data types, it employs an adaptive strategy determined by analyzing the first N lines (default $N = 1000$) of the input stream.

- **Strict Mode:** Captures quoted strings and explicit numbers. Ideal for **highly structured data with rigid syntax conventions**.
- **Aggressive Mode:** Captures alphanumeric tokens. Ideal for **semi-structured data or free-form text streams**.

Heuristic thresholds CAST performs a lightweight statistical analysis on the first N lines (default $N = 1000$) to select the parsing mode. In our implementation we use the following empirically chosen thresholds: if the ratio of unique skeletons to sample lines exceeds 0.10 we switch from **Strict** to **Aggressive** parsing. To avoid template explosion, the compressor enforces a template budget of $T_{max} = \alpha \cdot L$ (where L is number of lines and $\alpha = 0.25$ in Strict, $\alpha = 0.40$ in Aggressive); exceeding this budget triggers a safe passthrough (no structural transform).

Graceful Degradation The parsing mechanism is designed to handle schema drift without failure. If a line partially matches the **parsing rules**, the unmatched suffix is absorbed into the static skeleton rather than discarded. This ensures that irregular lines simply result in unique templates, allowing the algorithm to naturally degrade towards row-based compression behavior for noisy data, ensuring zero data loss.

2.2 Robustness: The Binary Guard

To prevent data corruption or inefficiency on non-textual files, CAST implements a "Binary Guard". Before processing, a 4KB sample is analyzed. If the density of non-printable control characters (excluding whitespace) exceeds 1%, the input is classified as binary. In this state, CAST enters a "Passthrough" mode, forwarding the raw byte stream directly to the backend compressor with zero structural modification.

3 Algorithm Detail

The core transformation logic executes a single-pass processing strategy (post-analysis) with a dynamic finalization step. The procedure is formally described in Algorithm 1.

Algorithm 1 CAST Compression Logic (overview)

```

1: Input: Byte stream  $D$ 
2: if IsLikelyBinary( $D$ ) then
3:   return BackendCompress( $D$ )                                 $\triangleright$  Passthrough (backend-agnostic)
4: end if
5: // Phase 1: Heuristic Analysis
6:  $Strategy \leftarrow \text{AnalyzeSample}(D)$                        $\triangleright$  choose Strict/Aggressive; sample  $N$  lines
7:  $Map \leftarrow \{\}, Skeletons \leftarrow [], Columns \leftarrow \{\}, StreamIDs \leftarrow []$ 
8: // Phase 2: Decomposition
9: for line in  $D$  do
10:   if ContainsReservedChars(line) then                                $\triangleright$  Fail-Safe: Collision detected
11:     return BackendCompress( $D$ )
12:   end if
13:    $S, V \leftarrow \text{Mask}(line, Strategy)$ 
14:   if  $S \notin Map$  then
15:     if templates_exceed_budget() then                                 $\triangleright$  Fallback: entropy limit
16:       return BackendCompress( $D$ )
17:     end if
18:      $Map[S] \leftarrow \text{NewID}()$ 
19:      $Skeletons.append(S)$ 
20:   end if
21:    $ID \leftarrow Map[S]$ 
22:    $Columns[ID].append(V)$ 
23:    $StreamIDs.append(ID)$ 
24: end for
// Phase 3: Unified / Split decision (heuristic)
25:  $B_{reg} \leftarrow \text{ToBytes}(Skeletons)$ 
26:  $B_{ids} \leftarrow \text{ToBytes}(StreamIDs)$ 
27:  $B_{vars} \leftarrow \text{ToBytes}(Columns)$ 
28:  $Blob \leftarrow \text{PackHeader}(B_{reg}, B_{ids}) + B_{vars}$ 
29: return BackendCompress( $Blob$ )                                      $\triangleright$  e.g., LZMA2, Zstd, Brotli
30: else
31:    $C_{reg} \leftarrow \text{BackendCompress}(\text{ToBytes}(Skeletons))$ 
32:    $C_{ids} \leftarrow \text{BackendCompress}(\text{ToBytes}(StreamIDs))$ 
33:    $C_{vars} \leftarrow \text{BackendCompress}(\text{ToBytes}(Columns))$ 
34:   return Package( $C_{reg}, C_{ids}, C_{vars}$ )
35: end if

```

Unified vs Split decision heuristic To balance compression density against memory usage, CAST employs a lightweight heuristic primarily based on template cardinality. For datasets with a limited set of unique structures ($\text{num_templates} < 256$), the algorithm **defaults to Unified Mode** to maximize the shared dictionary context of the backend compressor, **unless a preliminary sampling test detects poor compressibility**. Conversely, when template diversity is high, it strictly enforces *Split Mode* to improve parallelism and reduce the overhead

of managing a single monolithic context window.

Serialization format Skeletons are concatenated into a registry separated by the Unicode Private Use character U+E001 (selected to avoid collisions with standard text or Latin-1 binaries). Variables are organized per-column and serialized using a row separator (0x00) and a column separator (0x02).

To ensure bitwise reversibility and total binary safety, CAST applies a mandatory **Byte Stuffing** escape scheme regardless of the Unified/Split decision.

The escape byte 0x01 is used to encode reserved control characters (e.g., 0x00 becomes 0x01 0x00). This guarantees a collision-free stream even when processing mixed-encoding logs, binary artifacts, or multi-byte characters, eliminating the alignment failures associated with raw separation.

Conversely, **Split Mode** creates distinct physical streams to maximize parallelism, but adheres to the same escaped binary format to maintain data integrity. The decompressor performs the inverse unescaping prior to reconstruction.

Collision Safety & Fail-Safe Although the selected Private Use Area characters (U+E000, U+E001) are disjoint from standard text encodings and unreachable via Latin-1 binary decoding (which maps only up to U+00FF), CAST enforces a deterministic fail-safe mechanism. During the parsing phase, if the input stream is found to naturally contain these reserved markers, the algorithm aborts the transformation and transparently falls back to the backend compressor (Passthrough Mode). This guarantees that the structural reconstruction is never ambiguous, ensuring 100% data integrity even in the theoretical edge case of adversarial inputs.

Template ID encoding To minimize metadata overhead, template stream ids are encoded using one of four modes depending on the number of distinct templates:

- **Mode 3:** single template (no ids stored)
- **Mode 2:** 8-bit ids
- **Mode 0:** 16-bit ids
- **Mode 1:** 32-bit ids

The encoder selects the smallest-width representation that can store all template identifiers and records a compact id-mode flag in the header.

Integrity For unified (solid) compression CAST stores a compact header with lengths and relies on the underlying compressor integrity check (e.g., LZMA CRC32) and, optionally, an explicit CRC32 verification of the reconstructed payload to ensure bit-perfect lossless round-trip. To guarantee strictly lossless reconstruction (including mixed line-endings like CRLF/LF), the line parsing phase explicitly preserves original terminators.

4 Performance Evaluation

To validate the efficacy of CAST, we compiled a heterogeneous corpus of datasets sourced from **Kaggle**, **LogHub**, **Zenodo** and other public repositories. The dataset selection is **intentionally weighted** towards the algorithm’s target domain—structured machine-generated data—to fully explore the optimization potential in relevant scenarios.

However, to define the algorithm’s operational boundaries, we also included a small control group representing **low-redundancy scenarios** (including unstructured text and high-variance

structured files). This allows us to transparently verify the hypothesis that CAST’s benefits are strictly dependent on *exploitable* structural redundancy and to quantify the overhead when such redundancy is absent.

To fully evaluate the algorithm’s capabilities, we utilized two different **Rust implementations**. These implementations **feature** native **multi-threading** capabilities and configurable memory parameters (specifically `chunk-size` and `dict-size`) to strictly control the runtime footprint, preventing memory saturation and enabling the processing of datasets larger than available RAM.

We evaluated this engine in two specific configurations:

- **Native Mode:** Links directly against native libraries.
While this implementation fully supports multi-threading and configurable chunking, for the compression ratio analysis it was restricted to single-threaded, monolithic execution to strictly isolate the algorithmic efficiency of the structural transformation without the masking effects of parallelization or context fragmentation.
- **System Mode (7-Zip Backend):** Pipes data to the external `7z` executable.
This configuration leverages robust **multi-threading** for industrial scalability, achieving substantially higher speeds with negligible compression loss compared to the Native version.
Consequently, this benchmark scenario includes many additional large-scale datasets (e.g., 500 MB+) to stress-test the pipeline under heavy load conditions.

We benchmarked CAST against three state-of-the-art compression algorithms to provide a comprehensive landscape:

- **LZMA2 (XZ):** Preset 9 (Extreme) with a 128 MB **dictionary**. To guarantee a strictly fair comparison, we utilize the exact same engine as the CAST backend: the shared native library for the Native Mode tests, and the identical 7-Zip binary/arguments for the System Mode tests.
- **Zstandard (Zstd):** Level 22 (Ultra), representing modern high-performance compression.
- **Brotli:** Quality 11 (Max), widely used for web content optimization.

To ensure a comprehensive assessment, the benchmarks distinguish between three key performance metrics:

1. **Algorithmic Efficiency (Compression Ratio):** The reduction in file size compared to the original raw data, measured using the **Native Mode** to ensure binary-level precision.
2. **Compression Throughput:** We evaluate this primarily using the **System Mode** (7-Zip backend) to leverage an industry-standard, hyper-optimized encoding engine.
This approach demonstrates that CAST achieves **substantially** higher speeds by offloading the heavy lifting to a mature runtime, proving that top-tier performance is attainable without requiring a custom re-implementation of complex multi-threaded encoding logic.
3. **Restoration Latency (Decompression):** Unlike standard algorithms where decompression is a linear byte-stream inflation, CAST requires a *structural reconstruction phase*. We measure this overhead using **both configurations** to differentiate between the intrinsic algorithmic cost (Native Mode) and the end-to-end restoration time including external process management (System Mode).

4.1 Rust Native Benchmarks

Table 1 illustrates the reduction in file size and processing time across different algorithms using the Rust Native implementation.

Table 1: Rust Native Benchmark Results

Dataset	Original	LZMA2	Zstd	Brotli	CAST (ours)	Ratio
<i>CSV Datasets</i>						
Balance Payments	33.2 MB	501 KB (110s)	698 KB (103s)	592 KB (131s)	245 KB (4.7s)	139x
Migration Stats	29.3 MB	945 KB (56s)	1.12 MB (49s)	1.06 MB (87s)	319 KB (4.6s)	94x
Apple Sitemap	124.2 MB	2.21 MB (178s)	2.51 MB (218s)	2.50 MB (508s)	1.87 MB (34s)	66.4x
DDoS Data	616.8 MB	19.7 MB (1607s)	24.4 MB (1598s)	22.0 MB (1972s)	10.3 MB (417s)	60x
Subnat. Life Tables	16.0 MB	608 KB (13.8s)	713 KB (9.9s)	564 KB (52.5s)	324 KB (3.5s)	50.6x
COVID-19 Surveillance	872.1 MB	25.0 MB (1232s)	27.2 MB (800s)	27.1 MB (3386s)	20.3 MB (559s)	43x
Smart City Sensors	24.2 MB	1.02 MB (54s)	1.14 MB (47s)	1.05 MB (58s)	578 KB (3.6s)	43x
Train/Test Network	29.9 MB	1.05 MB (32.8s)	1.16 MB (26.2s)	1.10 MB (94.2s)	0.89 MB (6.6s)	33.6x
Wireshark	154.4 MB	9.52 MB (457s)	10.8 MB (302s)	10.1 MB (403s)	5.69 MB (167s)	27.1x
RT_IOT2022	54.8 MB	2.53 MB (144s)	2.53 MB (221s)	2.51 MB (47.4s)	1.99 MB (18.5s)	27.5x
NYC Bus Breakdowns	132.9 MB	9.79 MB (165s)	10.4 MB (127s)	10.9 MB (317s)	8.40 MB (93s)	15.8x
NZDep Life Tables	13.1 MB	1.16 MB (8.6s)	1.26 MB (6.5s)	1.14 MB (29s)	881 KB (2.8s)	15.2x
US Stock Prices	224.2 MB	26.9 MB (712s)	29.1 MB (521s)	27.9 MB (495s)	17.4 MB (140s)	13x
Covid Vaccinations	50.8 MB	4.65 MB (48s)	5.09 MB (39s)	4.58 MB (158s)	4.13 MB (21s)	12.3x
HomeC	131.0 MB	14.9 MB (257s)	15.6 MB (195s)	15.6 MB (330s)	11.2 MB (104s)	11.7x
Japan Trade 2020	207.9 MB	24.8 MB (463s)	26.4 MB (355s)	25.2 MB (511s)	18.4 MB (167s)	11.3x
IOT-temp	6.95 MB	788 KB (10.0s)	828 KB (8.6s)	797 KB (16.0s)	728 KB (5.2s)	9.8x
Aus/NZ Fires from space	73.0 MB	9.75 MB (93s)	10.7 MB (67.6s)	10.0 MB (224s)	7.69 MB (52.2s)	9.5x
HAI Security Train	114.2 MB	19.0 MB (123s)	19.6 MB (92s)	19.2 MB (246s)	13.0 MB (67s)	8.8x
IoT Intrusion	197.5 MB	25.1 MB (272s)	25.9 MB (195s)	28.0 MB (521s)	24.0 MB (135s)	8.2x
Nashville Housing	9.9 MB	1.41 MB (8.2s)	1.49 MB (5.9s)	1.41 MB (22s)	1.30 MB (5.3s)	7.6x
Owid Covid	46.7 MB	7.08 MB (55s)	7.49 MB (47s)	6.92 MB (126s)	6.39 MB (26s)	7.3x
Assaults 2015	234 KB	33.9 KB (0.16s)	37.6 KB (0.36s)	34.0 KB (0.41s)	39.6 KB (0.12s)	5.9x
ChatGPT Paraphrases	264.9 MB	40.1 MB (259s)	40.5 MB (200s)	44.4 MB (523s)	44.8 MB (262s)	5.9x
Item Aliases	201.5 MB	40.3 MB (436s)	43.6 MB (301s)	43.4 MB (370s)	40.2 MB (240s)	5x
PaySim Mobile Money	493.5 MB	148.8 MB (756s)	150.8 MB (578s)	154.9 MB (1103s)	130.1 MB (574s)	3.8x
Dielectron Collision	14.7 MB	5.71 MB (14.2s)	6.08 MB (12.0s)	5.88 MB (33s)	5.26 MB (12.6s)	2.8x
<i>Logs and SQL Datasets</i>						
OpenSSH Logs (LogHub)	70.02 MB	2.23 MB (126.8s)	2.84 MB (112.0s)	2.61 MB (187.3s)	1.01 MB (18.6s)	69.3x
PostgreSQL JSON Logs (Zenodo)	392.8 MB	9.29 MB (1674s)	10.5 MB (1434s)	10.4 MB (1560s)	6.20 MB (170s)	63.3x
Spider NLP Train	23.2 MB	482 KB (11.4s)	483 KB (15.6s)	509 KB (24.5s)	406 KB (3.4s)	57.1x
BGL (LogHub)	708.8 MB	26.6 MB (1330s)	31.2 MB (942s)	33.0 MB (2051s)	19.7 MB (242s)	36.0x
Sakila DB	8.8 MB	427 KB (26s)	502 KB (21s)	466 KB (28s)	298 KB (2.2s)	29.5x
Weblog Sample	67.6 MB	2.65 MB (67s)	2.88 MB (60s)	3.07 MB (204s)	2.53 MB (34s)	26.7x
Apache Server Logs	242.0 MB	12.9 MB (306s)	13.3 MB (217s)	14.2 MB (679s)	10.3 MB (131s)	23.5x
Cross Reference	12.0 MB	1.15 MB (15.7s)	1.28 MB (12.1s)	1.34 MB (26s)	1.08 MB (9.1s)	11.1x
World Cities SQL	20.3 MB	2.47 MB (22.5s)	2.78 MB (17.0s)	2.54 MB (48.2s)	2.48 MB (10.4s)	8.2x
Bible MySQL	32.3 MB	4.30 MB (39s)	4.45 MB (32s)	6.42 MB (69s)	4.30 MB (42s)	7.5x
mssql YLT Table SQL	12.8 MB	1.02 MB (30.9s)	1.15 MB (27.4s)	1.11 MB (31.8s)	1.02 MB (31.3s)	12.5x
Audit Dump (SQL)	64.6 MB	12.0 MB (135s)	12.6 MB (109s)	12.1 MB (146s)	10.2 MB (35s)	6.3x
Pixel 4XL GNSS Log	130.5 MB	32.1 MB (207s)	33.3 MB (150s)	31.5 MB (297s)	25.0 MB (219s)	5.2x
<i>JSON & XML Datasets</i>						
CERN OpenData Slim	211.9 MB	16.1 MB (393s)	17.0 MB (362s)	16.4 MB (510s)	15.2 MB (327s)	14x
Yelp Business	118.9 MB	9.87 MB (123s)	10.2 MB (102s)	11.2 MB (293s)	10.4 MB (73s)	11.4x
Brazil Geo	14.5 MB	1.55 MB (8.2s)	1.64 MB (6.1s)	1.67 MB (27s)	1.53 MB (9.8s)	9.5x
Wikidata Fanout	262.3 MB	33.0 MB (284s)	38.2 MB (205s)	34.8 MB (533s)	28.5 MB (181s)	9.2x
Pagerank	121.9 MB	14.8 MB (127s)	15.2 MB (136s)	16.5 MB (289s)	14.7 MB (129s)	8.3x
Yelp Tips	180.6 MB	34.2 MB (188s)	34.7 MB (132s)	44.1 MB (376s)	30.6 MB (107s)	5.9x
Yelp Checkin	287.0 MB	54.1 MB (500s)	61.9 MB (386s)	59.7 MB (866s)	53.6 MB (422s)	5.4x
Gandhi Works	100.6 MB	20.6 MB (106s)	20.9 MB (84s)	22.5 MB (217s)	20.3 MB (92s)	5x
Wiki 00c2bfc7.json	41.2 MB	10.26 MB (51s)	10.50 MB (41s)	10.80 MB (107s)	10.24 MB (49s)	4x
GloVe Emb.	193.4 MB	57.7 MB (400s)	57.9 MB (231s)	60.3 MB (501s)	57.1 MB (401s)	3.4x
<i>Text Datasets</i>						
XDados	4.4 MB	535 KB (3.1s)	597 KB (2.2s)	536 KB (9.7s)	420 KB (3.4s)	10.5x
Tatoeba French	26.9 MB	3.69 MB (24.5s)	3.87 MB (20.5s)	3.81 MB (51s)	5.14 MB (25s)	5.2x
Election Day Tweets	35.9 MB	10.1 MB (31s)	10.2 MB (24s)	10.4 MB (74s)	12.1 MB (38s)	3x

4.2 Rust + 7-Zip Benchmarks

A common drawback of pre-processing is added latency. However, Table 2 demonstrates that offloading the compression workload to the highly optimized 7-Zip backend not only eliminates the overhead observed in the Native implementation, but actually achieves higher throughput than applying 7-Zip directly to the raw file.

Controlled Environment & Analysis: To ensure a strictly fair comparison, the control group ("LZMA2 (7-Zip)") employs the exact same compression binary and threading configuration used by the CAST backend.

Consequently, the observed speedup is attributable solely to the entropy reduction achieved by the structural transformation.

Although CAST adds a parsing step, the resulting column-aligned streams (e.g., continuous integers or timestamps) are significantly less computationally expensive for the LZMA encoder to process than the raw chaotic text, resulting in a net reduction of total processing time.

Table 2: Rust + 7-Zip Benchmarks

Dataset	Original	LZMA2 (7-Zip)	CAST (Rust Native)	CAST (Rust+7-Zip)	Compression Ratio	Speedup (vs LZMA2)
<i>CSV Datasets</i>						
Balance Payments	33.1 MB	834 KB (2.0s)	245 KB (4.7s)	255 KB (0.75s)	138x	2.69x
Migration Stats	29.2 MB	1.38 MB (4.6s)	319 KB (4.6s)	343 KB (1.16s)	93.7x	4.00x
Sitemap Apple	124.2 MB	2.69 MB (9.25s)	1.87 MB (34s)	1.99 MB (6.51s)	66.4x	1.42x
Sowiport Rec Logs	2.30 GB	51.0 MB (189s)	-	39.3 MB (139s)	60x	1.36x
DDoS Data	616.8 MB	20.4 MB (81s)	10.3 MB (417s)	10.9 MB (37s)	60x	2.17x
Subnat. Life Tables	16.0 MB	824 KB (2.6s)	324 KB (3.5s)	344 KB (0.8s)	50.5x	3.27x
Smart City Sensors	23.04 MB	1.23 MB (5.8s)	578 KB (3.6s)	0.57 MB (1s)	40.4x	5.88x
COVID-19 Surveillance	872.1 MB	32.8 MB (63s)	-	22.3 MB (50s)	39x	1.26x
Train/Test Network	28.5 MB	1.29 MB (3.9s)	0.89 MB (6.6s)	0.89 MB (1.8s)	32x	2.14x
RT-IOT2022	54.8 MB	2.56 MB (8.7s)	1.99 MB (18.5s)	2.01 MB (5.9s)	27.5x	1.47x
Wireshark P3	154.4 MB	10.6 MB (47.7s)	5.69 MB (167s)	6.94 MB (32s)	27x	1.49x
Japan Trade 2018	668.3 MB	56.6 MB (105s)	-	25.9 MB (78s)	25.8x	1.33x
SQL Injection	1004.5 MB	60.1 MB (97s)	-	62.1 MB (182s)	16x	0.54x
NYC Bus Breakdowns	126.71 MB	10.55 MB (40.2s)	8.40 MB (93s)	8.24 MB (24.1s)	15.3x	1.67x
NZDep Life Tables	13.0 MB	1.20 MB (2.7s)	881 KB (2.8s)	882 KB (1.08s)	15x	2.53x
H1B Data	756.6 MB	45.5 MB (106s)	-	53.7 MB (95s)	14x	1.12x
US Stock Prices	213.82 MB	27.18 MB (94.1s)	17.4 MB (140s)	16.69 MB (48s)	12.8x	1.96x
HomeC	131.0 MB	15.4 MB (54.6s)	11.2 MB (104s)	11.7 MB (35.7s)	11.7x	1.53x
Japan Trade 2020	207.9 MB	25.3 MB (89s)	18.4 MB (167s)	19.0 MB (53.2s)	11.3x	1.68x
IOT Temp	6.9 MB	787 KB (1.45s)	728 KB (5.2s)	724 KB (0.94s)	9.7x	1.54x
Aus/NZ Fires from space	73.0 MB	9.74 MB (30.2s)	7.69 MB (52.2s)	7.76 MB (16.1s)	9.5x	1.87x
HAI Security Train	108.91 MB	18.27 MB (53.1s)	13.0 MB (67s)	12.4 MB (29.8s)	8.8x	1.78x
IoT Intrusion	197.5 MB	28.2 MB (99.7s)	24.0 MB (135s)	24.2 MB (63.5s)	8.2x	1.57x
Nashville Housing	9.9 MB	1.42 MB (2.4s)	1.30 MB (5.3s)	1.28 MB (1.7s)	7.7x	1.36x
OWID Covid	46.7 MB	7.20 MB (15.7s)	6.39 MB (26s)	6.36 MB (13.2s)	7.3x	1.19x
NASA Global Fire Data	502.2 MB	86.7 MB (177.5s)	-	72.7 MB (140.8s)	6.9x	1.26x
Assaults 2015	234 KB	34.4 KB (0.06s)	39.6 KB (0.12s)	39.9 KB (0.07s)	6.8x	0.86x
ChatGPT Paraphrases	252.6 MB	38.3 MB (142s)	44.8 MB (262s)	42.6 MB (151s)	5.9x	0.94x
Satellites Solar Wind	873.9 MB	211.5 MB (317.5s)	-	161.5 MB (214s)	5.4x	1.48x
Item Aliases	201.5 MB	40.6 MB (83.7s)	40.2 MB (240s)	40.1 MB (91.1s)	5x	0.92x
PaySim Mobile Money	470.6 MB	143.27 MB (283s)	130.1 MB (574s)	125.6 MB (264s)	3.7x	1.07x
Synthetic Financial Log	470.7 MB	143.3 MB (362s)	-	125.7 MB (263s)	3.7x	1.38x
Dielectron Collision	14.06 MB	5.44 MB (8.46s)	5.26 MB (12.6s)	4.99 MB (8.2s)	2.8x	1.03x
<i>Logs & SQL Datasets</i>						
OpenSSH Logs (LogHub)	70.02 MB	3.43 MB (11.2s)	-	1.01 MB (2.9s)	69.3x	3.84x
PostgreSQL JSON Logs (Zenodo)	392.83 MB	12.5 MB (36s)	6.2 MB (165s)	6.67 MB (25s)	63.3x	1.44x
IOTA logs 2.21 (merge)	1.48 GB	73.1 MB (83.6s)	-	40.8 MB (80.2s)	37.1x	1.04x
BGL (LogHub)	708.76 MB	31.49 MB (74.3s)	-	20.1 MB (46.5s)	35.2x	1.60x
Web Server Logs Labeled	2.66 GB	103.3 MB (245s)	-	86.0 MB (181s)	31.6x	1.36x
Sakila Insert	8.8 MB	492 KB (0.97s)	298 KB (2.2s)	297 KB (0.53s)	30.3x	1.83x
HDFS v1 (LogHub) *	1.47 GB	91.1 MB (194s)	-	49.8 MB (230s)	30.2x	0.84x
Apache Server Logs	242.0 MB	15.7 MB (52.6s)	10.3 MB (131s)	11.9 MB (33.2s)	23.5x	1.58x
Cross Reference	11.5 MB	1.11 MB (3.72s)	1.08 MB (9.1s)	1.04 MB (2.4s)	11x	1.52x
Bible MySQL	30.8 MB	4.16 MB (15s)	4.30 MB (42s)	4.16 MB (16s)	7.4x	0.94x
Audit Dump (SQL)	64.6 MB	12.4 MB (27.1s)	10.2 MB (35s)	10.0 MB (14.3s)	6.5x	1.90x
<i>JSON Datasets</i>						
CERN OpenData Slim	202.12 MB	16.66 MB (49.76s)	15.2 MB (327s)	15.42 MB (45.6s)	13.3x	1.09x
Yelp Business	118.9 MB	11.1 MB (32.5s)	10.4 MB (73s)	10.9 MB (23.6s)	11.4x	1.38x
Brazil Geo	14.5 MB	1.55 MB (2.5s)	1.53 MB (9.8s)	1.5 MB (3s)	9.6x	0.83x
Wikidata Fanout	262.3 MB	33.4 MB (139s)	28.5 MB (181s)	27.8 MB (119s)	9.4x	1.17x
Pagerank	121.9 MB	15.8 MB (45.4s)	14.7 MB (129s)	15.7 MB (48.6s)	8.3x	0.93x
Yelp Tips	180.6 MB	35.0 MB (79.3s)	30.6 MB (107s)	30.4 MB (53s)	5.9x	1.50x
Yelp Checkin	287.0 MB	55.0 MB (157s)	53.6 MB (422s)	54.2 MB (169s)	5.3x	0.93x
Gandhi Works	100.6 MB	20.8 MB (55.4s)	20.3 MB (92s)	20.3 MB (55.2s)	5x	1.00x
Wiki 00c2bfc7-.json	41.2 MB	10.3 MB (17.7s)	10.2 MB (49s)	10.2 MB (18.6s)	4x	0.95x
Glove Emb.	193.4 MB	58.1 MB (179s)	57.1 MB (401s)	57.8 MB (195s)	3.4x	0.92x
<i>Text Datasets</i>						
Russian Dictionary	269.2 MB	13.9 MB (109s)	-	13.9 MB (119s)	19.37x	0.91x
Xdados	4.4 MB	533 KB (1.1s)	420 KB (3.4s)	433 KB (1.26s)	10.7x	0.87x
Tatoeba French	26.9 MB	3.74 MB (8.1s)	5.14 MB (25s)	5.02 MB (9.3s)	5.36x	0.87x
Pixel 4XL GNSS Log	124.43 MB	30.87 MB (90.6s)	25.0 MB (219s)	24.3 MB (76.1s)	5.1x	1.19x
Election Day Tweets	34.26 MB	9.64 MB (13.4s)	12.1 MB (38s)	11.27 MB (17.3s)	3x	0.78x

* Configured with a chunk-size of 300 MB (vs default no chunking). Contrarily to typical behavior, this segmentation yielded superior compression compared to a monolithic (single-chunk) strategy, demonstrating that isolating local entropy in heterogeneous streams can significantly enhance backend efficiency.

4.3 Decompression and Reconstruction Analysis

CAST involves an additional *reconstruction phase*: the algorithm must re-interleave the decompressed columnar data back into the original row-based textual format. Table ?? compares the restoration time between the Rust Native implementation and the Rust implementation using the 7-Zip backend.

Analysis:

- **Reconstruction Overhead:** The results confirm that the structural reconstruction cost is negligible. The optimized Rust engine efficiently handles the re-interleaving process, performing restoration with minimal latency overhead.
- **Native vs. External Backend:** The System Mode (Rust + 7-Zip) generally outperforms the Native version on larger files due to 7-Zip’s highly optimized multi-threaded decoding engine. However, the Native implementation is competitive and occasionally faster on smaller datasets, likely due to the absence of process-spawning overhead.

Table 3: Decompression Time Comparison: Rust Native vs Rust + 7-Zip (Sorted by Speed)

Dataset	Original Size	Compressed Size (\approx)	Rust (Native)	Rust (+7-Zip)	Best Speed (MB/s)
Smart City Sensors	23.04 MB	0.57 MB	0.22s	0.20s	115.2
HDFS v1 (LogHub)	1.47 GB	50 MB	14.20s	14.10s	106.7
BGL (LogHub)	708.8 MB	20 MB	9.36s	8.20s	86.4
Recipes JSON	85.2 MB	6 MB	1.34s	1.04s	81.9
PostgreSQL JSON Logs (Zenodo)	392.8 MB	6.5 MB	5.35s	5.62s	73.4
US Stock Prices	213.82 MB	16.7 MB	3.90s	3.29s	65.0
CERN OpenData Slim	202.12 MB	15 MB	4.10s	3.21s	63.0
IOTA logs 2.21 (merge)	1.48 GB	41 MB	34.60s	26.17s	57.9
HAI Security Train	108.91 MB	12 MB	2.59s	1.93s	56.4
ChatGPT Paraphrases	252.6 MB	43 MB	5.67s	4.70s	53.7
Pixel 4XL GNSS Log	124.43 MB	24 MB	2.92s	2.33s	53.4
PaySim Mobile Money	470.6 MB	125 MB	13.76s	9.59s	49.1
NYC Bus Breakdowns	126.71 MB	8 MB	3.14s	3.25s	40.4
Election Day Tweets	34.26 MB	11 MB	2.11s	2.75s	16.2

Theoretical Complexity: Unlike iterative compression which requires costly pattern matching ($O(N \cdot W)$ where W is window size), the reconstruction phase is strictly linear $O(N)$. The decoder operates via direct memory copying of pre-calculated offsets, meaning throughput is bounded primarily by memory bandwidth rather than CPU cycles. This explains why the Rust implementation achieves restoration speeds comparable to raw I/O.

5 Limitations & Constraints

1. **Binary & High Entropy Data:** As evidenced by benchmarks, CAST provides no benefit for binary/high entropy data. The Binary Guard correctly identifies these files, resulting in performance identical to standard LZMA2 (Equal size/time), but no gain.
2. **Vector Data:** On datasets consisting primarily of high-variance floating point numbers (e.g., GloVe Embeddings), the structural overhead matches the compression gain, resulting in a ratio gain of only 1.00x.
3. **Small Files:** For files under 1MB, the overhead of the CAST header and dictionary structure may result in a slightly larger file size compared to raw compression.

4. **Implementation Maturity (PoC):** The implementations presented are designed as scientific proofs-of-concept.

While functional and quite robust, they lack the extensive error handling, fuzz-testing, and security auditing required for deployment in mission-critical production environments.

Future engineering efforts will focus on hardening the codebase against malformed inputs and optimizing memory safety for edge cases.

6 Related Work

CAST intersects with several areas in data representation and compression. Here we position our contribution relative to prior art, specifically distinguishing between analytic-focused storage and archival compression.

Columnar storage Formats such as Apache Parquet and ORC store data column-wise to optimize analytic workloads (OLAP). However, they generally require explicit schemas definition upfront and are widely adopted for structured data lakes rather than raw log archival. CAST aims to bring similar entropy reduction benefits to schema-less, textual inputs without enforcing strict typing or migration to a new storage format.

Log-specific and Structural Compressors State-of-the-art tools like CLP, Crystal, and frameworks like OpenZL operate on principles similar to CAST: they perform structural decomposition (separating static templates from dynamic variables) without requiring an upfront manually defined schema. However, a key distinction lies in the design goal.

Systems like CLP and Crystal are **tailored specifically for log data** and often prioritize *searchability* and granular random access (e.g., query execution directly on compressed blocks). In contrast, CAST is designed primarily for *cold storage* and maximum archival density. It acts as a pure transformation layer to feed strong backend compressors (like LZMA2), prioritizing reduction ratio over immediate query latency.

While CAST currently utilizes chunking for memory safety, this block-based architecture shares the potential for future indexed random access implementations, similar to the aforementioned systems.

Preconditioning and Transform Filters Transforms such as Delta encoding, Burrows–Wheeler Transform (BWT), and dictionary pre-filters are commonly used as preprocessing steps to aid compression. CAST fits into this category but targets *structural* redundancy (syntax and hierarchy) rather than numeric locality or symbol re-ordering, making it orthogonal and complementary to these filters.

Summary CAST does not seek to replace analytic formats or search-optimized log compressors. Instead, it serves as a lightweight, reversible structural normalization layer. It is particularly effective for long-term archival scenarios where the primary metric is storage efficiency, bridging the gap between raw text compression and specialized structural encoding.

7 Reproducibility

Source code and benchmarking scripts are available in the project repository.

To facilitate immediate adoption, the tool requires minimal setup and features a streamlined command-line interface (CLI) designed for intuitive usage—analogous to standard utilities like 7-Zip.

To ensure reproducibility without distributing large third-party files, the datasets used are all

publicly available, with their specific sources cited directly in this paper and listed in the repository documentation.

Benchmarks were executed on a commodity workstation running **Windows 10 Pro for Workstations** (Build 19045), equipped with a 6th-generation **Intel Core i7 CPU** (Skylake architecture, 3.40 GHz) and 16 GB of physical RAM. Experiments were conducted using native Rust builds (`rustc 1.92.0`), Python 3.10 and 7-Zip 23.01.

Exact invocation commands are documented in the repository.

8 Conclusion

This work demonstrates that structural pre-processing can significantly enhance the effectiveness of general-purpose compression algorithms when applied to structured text data. By dynamically inferring and separating static templates from variable fields, CAST reshapes row-oriented text into column-aligned streams that better expose redundancy to standard compressors.

While CAST does not provide benefits for binary or high-entropy data, empirical results show substantial improvements in both compression ratio and throughput for logs, CSV files, and semi-structured datasets. These findings indicate that schema-free structural normalization represents a practical and lightweight approach for improving archival compression pipelines without altering or replacing existing compression backends.

Future Work. Finally, a promising direction lies in leveraging CAST’s block-based architecture to enable indexed random access. While the current chunking mechanism focuses on memory safety, it lays the foundation for granular retrieval and searchability directly on compressed segments, potentially bridging the gap between deep archival efficiency and query-ready storage.

Availability

The source code and the benchmarking harness are available under the MIT License at:
<https://github.com/AndreaLVR/CAST>