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 CSV files, JSON documents, XML, 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 unsuitable 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-allocation parser).
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 drastic 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:

$$L \to S + V$$

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 light-weight 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 4 KB 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($D$)                          ▷ Passthrough (backend-agnostic)
 4: end if
    // Phase 1: Heuristic Analysis
 5: $Strategy \leftarrow$ AnalyzeSample($D$)        ▷ choose Strict/Aggressive; sample $N$ lines
 6: $Map \leftarrow \{\}$, $Skeletons \leftarrow []$, $Columns \leftarrow \{\}$, $StreamIDs \leftarrow []$
    // Phase 2: Decomposition
 7: for line in $D$ do
 8:     if ContainsReservedChars(line) then                 ▷ Fail-Safe: Collision detected
 9:         return BackendCompress($D$)
10:     end if
11:     $S, V \leftarrow$ Mask($line, Strategy$)
12:     if $S \notin Map$ then
13:         if templates_exceed_budget() then
14:             return BackendCompress($D$)                  ▷ Fallback: entropy limit
15:         end if
16:         $Map[S] \leftarrow$ NewID()
17:         $Skeletons$.append($S$)
18:     end if
19:     $ID \leftarrow Map[S]$
20:     $Columns[ID]$.append($V$)
21:     $StreamIDs$.append($ID$)
22: end for
    // Phase 3: Unified / Split decision (heuristic)
23: $Decision \leftarrow$ DecideUnifiedOrSplit($Columns, Skeletons$)
24: if $Decision$ is Unified then
25:     $B_{reg} \leftarrow$ ToBytes($Skeletons$)
26:     $B_{ids} \leftarrow$ ToBytes($StreamIDs$)
27:     $B_{vars} \leftarrow$ ToBytes($Columns$)
28:     $Blob \leftarrow$ PackHeader($B_{reg}, B_{ids}$) + $B_{vars}$
29:     return BackendCompress(Blob)                         ▷ e.g., LZMA2, Zstd, Brotli
30: else
31:     $C_{reg} \leftarrow$ BackendCompress(ToBytes($Skeletons$))
32:     $C_{ids} \leftarrow$ BackendCompress(ToBytes($StreamIDs$))
33:     $C_{vars} \leftarrow$ BackendCompress(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 (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** and 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 a configurable `chunk-size` parameter used to split large input files into manageable segments, 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 significantly higher speeds with negligible compression loss compared to the Native version. **Consequently, this benchmark scenario includes 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:** The speed at which the algorithm transforms and writes data. We evaluate this primarily using the **System Mode** (7-Zip backend) to simulate a realistic production pipeline involving parallelized external tools.
   This configuration demonstrates significantly **higher speeds with negligible compression loss** compared to the Native version.
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)**
			CSV Datasets		
Balance Payments	33.2 MB	501 KB (110s)	698 KB (103s)	592 KB (131s)	**245 KB (4.7s)**
Migration Stats	29.3 MB	945 KB (56s)	1.12 MB (49s)	1.06 MB (87s)	**319 KB (4.6s)**
Subnat. Life Tables	16.0 MB	608 KB (13.8s)	713 KB (9.9s)	564 KB (52.5s)	**324 KB (3.5s)**
NZDep Life Tables	13.1 MB	1.16 MB (8.6s)	1.26 MB (6.5s)	1.14 MB (29s)	**881 KB (2.8s)**
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)**
Japan Trade 2020	207.9 MB	24.8 MB (463s)	26.4 MB (355s)	25.2 MB (511s)	**18.4 MB (167s)**
US Stock Prices	224.2 MB	26.9 MB (712s)	29.1 MB (521s)	27.9 MB (495s)	**17.4 MB (140s)**
COVID-19 Surveillance	872.1 MB	25.0 MB (1232s)	27.2 MB (800s)	27.1 MB (3386s)	**20.3 MB (559s)**
PaySim Mobile Money	493.5 MB	148.8 MB (756s)	150.8 MB (578s)	154.9 MB (1103s)	**130.1 MB (574s)**
Covid Vaccinations	50.8 MB	4.65 MB (48s)	5.09 MB (39s)	4.58 MB (158s)	**4.13 MB (21s)**
HAI Security Train	114.2 MB	19.0 MB (123s)	19.6 MB (92s)	19.2 MB (246s)	**13.0 MB (67s)**
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)**
NYC Bus Breakdowns	132.9 MB	9.79 MB (165s)	10.4 MB (127s)	10.9 MB (317s)	**8.40 MB (93s)**
IOT-temp	6.95 MB	788 KB (10.0s)	828 KB (8.6s)	797 KB (16.0s)	**728 KB (5.2s)**
Apple Sitemap	124.2 MB	2.21 MB (178s)	2.51 MB (218s)	2.50 MB (508s)	**1.87 MB (34s)**
Nashville Housing	9.9 MB	1.41 MB (8.2s)	1.49 MB (5.9s)	1.41 MB (22s)	**1.30 MB (5.3s)**
Item Aliases	201.5 MB	40.3 MB (436s)	43.6 MB (301s)	43.4 MB (370s)	**40.2 MB (240s)**
IoT Intrusion	197.5 MB	25.1 MB (272s)	25.9 MB (195s)	28.0 MB (521s)	**24.0 MB (135s)**
HomeC	131.0 MB	14.9 MB (257s)	15.6 MB (195s)	15.6 MB (330s)	**11.2 MB (104s)**
DDoS Data	616.8 MB	19.7 MB (1607s)	24.4 MB (1598s)	22.0 MB (1972s)	**10.3 MB (417s)**
Wireshark	154.4 MB	9.52 MB (457s)	10.8 MB (302s)	10.1 MB (403s)	**5.69 MB (167s)**
RT_IOT2022	54.8 MB	2.53 MB (144s)	2.53 MB (221s)	2.51 MB (47.4s)	**1.99 MB (18.5s)**
Owid Covid	46.7 MB	7.08 MB (55s)	7.49 MB (47s)	6.92 MB (126s)	**6.39 MB (26s)**
Assaults 2015	234 KB	**33.9 KB** (0.16s)	37.6 KB (0.36s)	34.0 KB (0.41s)	39.6 KB (**0.12s**)
ChatGPT Paraphrases	264.9 MB	**40.1 MB** (259s)	40.5 MB (**200s**)	44.4 MB (523s)	44.8 MB (262s)
Dielectron Collision	14.7 MB	5.71 MB (14.2s)	6.08 MB (**12.0s**)	5.88 MB (33s)	**5.26 MB** (12.6s)
Smart City Sensors	24.2 MB	1.02 MB (54s)	1.14 MB (47s)	1.05 MB (58s)	**578 KB (3.6s)**
			JSON & XML Datasets		
Wikidata Fanout	262.3 MB	33.0 MB (284s)	38.2 MB (205s)	34.8 MB (533s)	**28.5 MB (181s)**
Gandhi Works	100.6 MB	20.6 MB (106s)	20.9 MB (**84s**)	22.5 MB (217s)	**20.3 MB** (92s)
Yelp Business	118.9 MB	**9.87 MB** (123s)	10.2 MB (102s)	11.2 MB (293s)	10.4 MB (**73s**)
Yelp Tips	180.6 MB	34.2 MB (188s)	34.7 MB (132s)	44.1 MB (376s)	**30.6 MB (107s)**
Yelp Checkin	287.0 MB	54.1 MB (500s)	61.9 MB (**386s**)	59.7 MB (866s)	**53.6 MB** (422s)
Wiki 00c2bfc7-.json	41.2 MB	10.26 MB (51s)	10.50 MB (**41s**)	10.80 MB (107s)	**10.24 MB** (49s)
GloVe Emb.	193.4 MB	57.7 MB (400s)	57.9 MB (**231s**)	60.3 MB (501s)	**57.1 MB** (401s)
Pagerank	121.9 MB	14.8 MB (**127s**)	15.2 MB (136s)	16.5 MB (289s)	**14.7 MB** (129s)
Brazil Geo	14.5 MB	1.55 MB (8.2s)	1.64 MB (**6.1s**)	1.67 MB (27s)	**1.53 MB** (9.8s)
CERN OpenData Slim	211.9 MB	16.1 MB (393s)	17.0 MB (362s)	16.4 MB (510s)	**15.2 MB (327s)**
			Logs and SQL Datasets		
Logfiles	242.0 MB	12.9 MB (306s)	13.3 MB (217s)	14.2 MB (679s)	**10.3 MB (131s)**
Weblog Sample	67.6 MB	2.65 MB (67s)	2.88 MB (60s)	3.07 MB (204s)	**2.53 MB (34s)**
Audit Dump (SQL)	64.6 MB	12.0 MB (135s)	12.6 MB (109s)	12.1 MB (146s)	**10.2 MB (35s)**
Sakila DB	8.8 MB	427 KB (26s)	502 KB (21s)	466 KB (28s)	**298 KB (2.2s)**
Pixel 4XL GNSS Log	130.5 MB	32.1 MB (207s)	33.3 MB (**150s**)	31.5 MB (297s)	**25.0 MB** (219s)
Spider NLP Train	23.2 MB	482 KB (11.4s)	483 KB (15.6s)	509 KB (24.5s)	**406 KB (3.4s)**
Cross Reference	12.0 MB	1.15 MB (15.7s)	1.28 MB (12.1s)	1.34 MB (26s)	**1.08 MB (9.1s)**
Bible MySQL	32.3 MB	**4.30 MB** (39s)	4.45 MB (**32s**)	6.42 MB (69s)	4.30 MB (42s)
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**)
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)
			Text Datasets		
Tatoeba French	26.9 MB	**3.69 MB** (24.5s)	3.87 MB (**20.5s**)	3.81 MB (51s)	5.14 MB (25s)
Election Day Tweets	35.9 MB	**10.1 MB** (31s)	10.2 MB (**24s**)	10.4 MB (74s)	12.1 MB (38s)
XDados	4.4 MB	535 KB (3.1s)	597 KB (**2.2s**)	536 KB (9.7s)	**420 KB** (3.4s)

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)	Speedup (vs LZMA2)
CSV Datasets					
Balance Payments	33.1 MB	834 KB (2.02s)	**245 KB** (4.7s)	255 KB (**0.75s**)	**2.69x**
Migration Stats	29.2 MB	1.38 MB (4.64s)	**319 KB** (4.6s)	343 KB (**1.16s**)	**4.00x**
Subnat. Life Tables	16.0 MB	824 KB (2.65s)	**324 KB** (3.5s)	344 KB (**0.81s**)	**3.27x**
NZDep Life Tables	13.0 MB	1.20 MB (2.73s)	**881 KB** (2.8s)	882 KB (**1.08s**)	**2.53x**
Satellites Solar Wind	873.9 MB	211.5 MB (317.52s)	-	**161.5 MB (214s)**	**1.48x**
Sowiport Rec Logs	2.30 GB	51.0 MB (189.6s)	-	**39.3 MB (139.5s)**	**1.36x**
COVID-19 Surveillance	872.1 MB	32.8 MB (63.5s)	-	**22.3 MB (50.4s)**	**1.26x**
Aus/NZ Fires from space	73.0 MB	9.74 MB (30.26s)	**7.69 MB** (52.2s)	7.76 MB (**16.19s**)	**1.87x**
Japan Trade 2020	207.9 MB	25.3 MB (89.4s)	**18.4 MB** (167s)	19.0 MB (**53.2s**)	**1.68x**
Japan Trade 2018	668.3 MB	56.6 MB (105s)	-	**25.9 MB (78.7s)**	**1.33x**
US Stock Prices	213.82 MB	27.18 MB (94.09s)	**17.4 MB** (140s)	16.69 MB (**48s**)	**1.96x**
PaySim Mobile Money	470.6 MB	143.27 MB (283s)	130.1 MB (574s)	**125.6 MB (264s)**	**1.07x**
NASA Global Fire Data	502.2 MB	86.7 MB (177.53s)	-	**72.7 MB (140.84s)**	**1.26x**
HAI Security Train	108.91 MB	18.27 MB (53.1s)	13.0 MB (67s)	**12.4 MB (29.8s)**	**1.78x**
Synthetic Financial Log	470.7 MB	143.3 MB (362.4s)	-	**125.7 MB (263.1s)**	**1.38x**
Train/Test Network	28.5 MB	1.29 MB (3.89s)	**0.89 MB** (6.6s)	**0.89 MB (1.82s)**	**2.14x**
NYC Bus Breakdowns	126.71 MB	10.55 MB (40.2s)	**8.40 MB** (93s)	8.24 MB (**24.1s**)	**1.67x**
IOT Temp	6.9 MB	787 KB (1.45s)	728 KB (5.2s)	**724 KB (0.94s)**	**1.54x**
Sitemap Apple	124.2 MB	2.69 MB (9.25s)	**1.87 MB** (34s)	1.99 MB (**6.51s**)	**1.42x**
Nashville Housing	9.9 MB	1.42 MB (2.36s)	1.30 MB (5.3s)	**1.28 MB (1.74s)**	**1.36x**
Item Aliases	201.5 MB	40.6 MB (**83.7s**)	40.2 MB (240s)	**40.1 MB** (91.1s)	0.92x
IoT Intrusion	197.5 MB	28.2 MB (99.7s)	**24.0 MB** (135s)	24.2 MB (**63.5s**)	**1.57x**
HomeC	131.0 MB	15.4 MB (54.6s)	**11.2 MB** (104s)	11.7 MB (**35.7s**)	**1.53x**
DDoS Data	616.8 MB	20.4 MB (81.1s)	**10.3 MB** (417s)	10.9 MB (**37.3s**)	**2.17x**
Wireshark P3	154.4 MB	10.6 MB (47.7s)	**5.69 MB** (167s)	6.94 MB (**32.1s**)	**1.49x**
RT_IOT2022	54.8 MB	2.56 MB (8.66s)	**1.99 MB** (18.5s)	2.01 MB (**5.9s**)	**1.47x**
OWID Covid	46.7 MB	7.20 MB (15.7s)	6.39 MB (26s)	**6.36 MB (13.2s)**	**1.19x**
Assaults 2015	234 KB	**34.4 KB (0.06s)**	39.6 KB (0.12s)	39.9 KB (0.07s)	0.86x
ChatGPT Paraphrases	252.6 MB	**38.3 MB (142.4s)**	44.8 MB (262s)	42.6 MB (151.3s)	0.94x
Dielectron Collision	14.06 MB	5.44 MB (8.46s)	5.26 MB (12.6s)	**4.99 MB (8.20s)**	**1.03x**
Smart City Sensors	23.04 MB	1.23 MB (5.88s)	578 KB (3.6s)	**0.57 MB (1.0s)**	**5.88x**
H1B Data	756.6 MB	**45.5 MB** (106.1s)	-	53.7 MB (**94.9s**)	**1.12x**
SQL Injection	1004.5 MB	**60.1 MB (97.4s)**	-	62.1 MB (182s)	0.54x
JSON Datasets					
Wikidata Fanout	262.3 MB	33.4 MB (139s)	28.5 MB (181s)	**27.8 MB (119s)**	**1.17x**
Gandhi Works	100.6 MB	20.8 MB (55.4s)	**20.3 MB** (92s)	**20.3 MB (55.2s)**	**1.00x**
Yelp Business	118.9 MB	11.1 MB (32.5s)	**10.4 MB** (73s)	10.9 MB (**23.6s**)	**1.38x**
Yelp Tips	180.6 MB	35.0 MB (79.3s)	30.6 MB (107s)	**30.4 MB (53s)**	**1.50x**
Yelp Checkin	287.0 MB	55.0 MB (**157s**)	**53.6 MB** (422s)	54.2 MB (169s)	0.93x
Wiki 00c2bfc7-.json	41.2 MB	10.3 MB (**17.7s**)	**10.2 MB** (49s)	**10.2 MB** (18.6s)	0.95x
Glove Emb.	193.4 MB	58.1 MB (**179s**)	**57.1 MB** (401s)	57.8 MB (195s)	0.92x
Pagerank	121.9 MB	15.8 MB (**45.4s**)	**14.7 MB** (129s)	15.7 MB (48.6s)	0.93x
Brazil Geo	14.5 MB	1.55 MB (**2.52s**)	1.53 MB (9.8s)	**1.5 MB** (3.03s)	0.83x
CERN OpenData Slim	202.12 MB	16.66 MB (49.76s)	**15.2 MB** (327s)	15.42 MB (**45.69s**)	**1.09x**
Logs & SQL Datasets					
IOTA logs 2.21 (merge)	1.48 GB	73.1 MB (83.6s)	-	**40.8 MB (80.2s)**	**1.04x**
Web Server Logs Labeled	2.66 GB	103.3 MB (245.4s)	-	**86.0 MB (181.0s)**	**1.36x**
Logfiles	242.0 MB	15.7 MB (52.6s)	**10.3 MB** (131s)	11.9 MB (**33.2s**)	**1.58x**
Weblog Sample	67.6 MB	3.16 MB (9.09s)	**2.53 MB** (34s)	2.90 MB (**7.7s**)	**1.18x**
Audit Dump (SQL)	64.6 MB	12.4 MB (27.1s)	10.2 MB (35s)	**10.0 MB (14.3s)**	**1.90x**
Sakila Insert	8.8 MB	492 KB (0.97s)	298 KB (2.2s)	**297 KB (0.53s)**	**1.83x**
Cross Reference	11.5 MB	1.11 MB (3.72s)	1.08 MB (9.1s)	**1.04 MB (2.44s)**	**1.52x**
Bible MySQL	30.8 MB	**4.16 MB (15.1s)**	4.30 MB (42s)	**4.16 MB** (16.02s)	0.94x
Text Datasets					
Tatoeba French	26.9 MB	**3.74 MB (8.09s)**	5.14 MB (25s)	5.02 MB (9.3s)	0.87x
Election Day Tweets	34.26 MB	**9.64 MB (13.44s)**	12.1 MB (38s)	11.27 MB (17.3s)	0.78x
Pixel 4XL GNSS Log	124.43 MB	30.87 MB (90.66s)	25.0 MB (219s)	**24.3 MB (76.13s)**	**1.19x**
Xdados	4.4 MB	533 KB (**1.10s**)	**420 KB** (3.4s)	433 KB (1.26s)	0.87x
Russian Dictionary	269.2 MB	**13.9 MB (108.9s)**	-	13.9 MB (119.4s)	0.91x

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 3 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

Dataset	**Original Size**	**Compressed Size (≈)**	**Rust (Native)**	**Rust (+7-Zip)**
		CSV Datasets		
US Stock Prices	213.82 MB	16.7 MB	3.90s	**3.29s**
PaySim Mobile Money	470.6 MB	125 MB	13.76s	**9.59s**
HAI Security Train	108.91 MB	12 MB	2.59s	**1.93s**
NYC Bus Breakdowns	126.71 MB	8 MB	**3.14s**	3.25s
ChatGPT Paraphrases	252.6 MB	43 MB	5.67s	**4.70s**
Dielectron Collision	14.06 MB	5 MB	**0.45s**	1.20s
Smart City Sensors	23.04 MB	0.57 MB	0.22s	**0.20s**
Item Aliases	201.5 MB	40 MB	**5.48s**	6.01s
		JSON Datasets		
CERN OpenData Slim	202.12 MB	15 MB	4.10s	**3.21s**
Recipes JSON	85.2 MB	6 MB	1.34s	**1.04s**
		Logs & SQL Datasets		
IOTA logs 2.21 (merge)	1.48 GB	41 MB	34.60s	**26.17s**
		Text Datasets		
Election Day Tweets	34.26 MB	11 MB	**2.11s**	2.75s
Pixel 4XL GNSS Log	124.43 MB	24 MB	2.92s	**2.33s**

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 the contribution relative to prior art.

Columnar storage Formats such as Apache Parquet and ORC store data column-wise and apply per-column encodings and compression; they require explicit schemas and are optimized for analytic workloads. CAST aims to bring columnar-like entropy reduction to schema-less, textual inputs without requiring an upfront schema or a separate storage format.

Grammar- and dictionary-based compressors Grammar-based compressors (e.g., Sequitur, RePair) and enhanced dictionary approaches build global structures representing repeated substrings or phrases. These approaches are related in spirit but differ: CAST focuses on structural decomposition (skeletons vs. variables) and explicit columnar reassembly rather than general grammar induction.

Preconditioning and transform filters Transforms such as Delta encoding, Burrows–Wheeler Transform (BWT), and dictionary pre-filters are commonly used as preprocessing steps. CAST is another form of preconditioning that targets structural redundancy instead of numeric locality or symbol re-ordering.

XML/JSON specific compressors Specialized compressors and encodings for XML (e.g., EXI) and JSON exploit known grammars or schemas. CAST differs by being *schema-free* and by attempting runtime inference for a broad class of structured text.

Summary CAST does not replace these techniques but complements them: it is a lightweight, reversible structural normalization layer intended to be stacked before a conventional backend compressor.

7 Reproducibility

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

Availability

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