CAST: Columnar Agnostic Structural Transformation

A Schema-less Structural Preprocessing Technique for Improving General-Purpose Compression

Andrea Olivari

December 2025

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, its robustness mechanisms (Binary Guard), and evaluates its effectiveness along two axes: maximum compression density (via a Python reference implementation) and practical throughput (via a Rust performance-oriented implementation).

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 regular-expression based engine to parse 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. Best for JSON/XML.
- **Aggressive Mode:** Captures alphanumeric tokens. Best for log files with variable syntax.

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).

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:     $S, V \leftarrow$ Mask($line, Strategy$)
9:     if $S \notin Map$ then
10:        if templates_exceed_budget() then
11:            return BackendCompress($D$)                  ▷ Fallback: entropy limit
12:        end if
13:        $Map[S] \leftarrow$ NewID()
14:        $Skeletons$.append($S$)
15:    end if
16:    $ID \leftarrow Map[S]$
17:    $Columns[ID]$.append($V$)
18:    $StreamIDs$.append($ID$)
19: end for
   // Phase 3: Unified / Split decision (heuristic)
20: $Decision \leftarrow$ DecideUnifiedOrSplit($Columns, Skeletons$)
21: if $Decision$ is Unified then
22:     $B_{reg} \leftarrow$ ToBytes($Skeletons$)
23:     $B_{ids} \leftarrow$ ToBytes($StreamIDs$)
24:     $B_{vars} \leftarrow$ ToBytes($Columns$)
25:     $Blob \leftarrow$ PackHeader($B_{reg}, B_{ids}$) + $B_{vars}$
26:     return BackendCompress(Blob)                       ▷ e.g., LZMA2, Zstd, Brotli
27: else
28:     $C_{reg} \leftarrow$ BackendCompress(ToBytes($Skeletons$))
29:     $C_{ids} \leftarrow$ BackendCompress(ToBytes($StreamIDs$))
30:     $C_{vars} \leftarrow$ BackendCompress(ToBytes($Columns$))
31:     return Package($C_{reg}, C_{ids}, C_{vars}$)
32: end if
```

Unified vs Split decision heuristic For medium-sized template sets (num_templates < 256) CAST computes a small sample of column bytes and compresses it with a fast zlib pass. If the ratio raw_sample_size/zlib(sample_size) is below an empirical threshold (3.0 in our experiments), the algorithm selects *Split Mode* to avoid poor dictionary utilization; otherwise it selects *Unified Mode* to maximize shared LZ context. This heuristic is lightweight and intended to choose the best trade-off between global dictionary reuse and parallelism/low-memory operation.

Serialization format Skeletons are concatenated into a registry separated by the unambiguous control byte `0x1E` (ASCII `RS`). Variables are organized per-column and serialized using a row separator (`0x00`) and a column separator (unified mode uses `0x02`, split mode uses the two-byte marker `0xFF 0xFF`).
To ensure bitwise reversibility in **Unified Mode**, CAST applies a byte-stuffing escape scheme: escape byte `0x01` is doubled (`0x01 0x01`), row separator is escaped as `0x01 0x00`, and the unified column separator (`0x02`) as `0x01 0x03`.

Conversely, **Split Mode** prioritizes throughput by bypassing the escape layer, utilizing the marker `0xFF 0xFF`.
Since CAST transcodes all variables to UTF-8 for storage (where the byte `0xFF` is invalid), this separator ensures deterministic collision avoidance for textual data. The decompressor performs the inverse unescaping prior to reconstruction.

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 across different domains, we compiled a heterogeneous corpus of datasets sourced from **Kaggle** and public open-data repositories (e.g., Stats NZ). The selection includes financial records, IoT telemetry, server logs, and linguistic corpora, ensuring the algorithm is tested against a wide spectrum of structural entropy profiles.
To fully evaluate both the theoretical potential and practical viability of the algorithm, we utilized two distinct implementations:

- **Python Reference Implementation:** Designed to measure the maximum theoretical compression density (Compression Ratio). This version serves as a baseline and does not include low-level optimizations; it operates single-threaded and processes files as monolithic blocks.
- **Rust Performance Implementation:** Designed for production-grade simulation throughput. It includes optimizations such as **multi-threading** for parallel processing and a configurable `chunk-size` parameter. This feature splits large input files into manageable segments, preventing memory saturation and enabling the processing of datasets larger than available RAM.

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

- **LZMA2 (XZ):** Preset 9 (Extreme), representing high-ratio archival compression.
- **Zstandard (Zstd):** Level 22 (Ultra), representing modern high-performance compression.
- **Brotli:** Quality 11 (Max), widely used for web content.

To ensure a fair comparison, the benchmarks distinguish between algorithmic efficiency (Compression Ratio) and implementation capability (Speed).

4.1 Compression Ratio (Python Reference)

Table 1 illustrates the reduction in file size and processing time across different algorithms. The Python implementation was used to measure the theoretical maximum density.

Table 1: Full Compression Benchmark (Size & Time)

Dataset	**Original**	**LZMA2**	**Zstd**	**Brotli**	**CAST (Ref)**
			CSV Datasets		
Balance Payments	33.1 MB	501 KB (94s)	697 KB (100s)	590 KB (90s)	**244 KB (5.5s)**
Migration Stats	29.2 MB	945 KB (49s)	1.12 MB (47s)	1.05 MB (68s)	**317 KB (6.9s)**
DDoS Data	616.7 MB	19.6 MB (1308s)	24.3 MB (1371s)	21.9 MB (1490s)	**10.2 MB (463s)**
RT_IOT2022	54.8 MB	2.53 MB (141s)	2.53 MB (240s)	2.51 MB (40s)	**1.99 MB (23.5s)**
Wireshark	154.4 MB	9.5 MB (312s)	10.7 MB (314s)	10.1 MB (325s)	**5.8 MB (145s)**
Metasploitable	52.8 MB	3.7 MB (**23s**)	3.8 MB (47s)	3.7 MB (126s)	**3.5 MB (28s)**
HomeC	131.0 MB	14.8 MB (189s)	15.5 MB (184s)	15.5 MB (266s)	**11.1 MB (103s)**
Custom 2020	207.9 MB	24.7 MB (449s)	26.4 MB (448s)	25.1 MB (478s)	**18.4 MB (213s)**
Owid Covid	46.7 MB	7.1 MB (49s)	7.5 MB (50s)	6.9 MB (112s)	**6.3 MB (29s)**
Gafgyt Botnet	105.8 MB	25.7 MB (161s)	25.6 MB (174s)	24.6 MB (237s)	**22.6 MB (128s)**
Assaults 2015	234 KB	**33.9 KB (0.1s)**	37.6 KB (0.3s)	34.0 KB (0.4s)	39.5 KB (0.2s)
			JSON & XML Datasets		
Votes Archive (XML)	145.8 MB	4.7 MB (192s)	5.6 MB (183s)	5.4 MB (285s)	**3.6 MB (57s)**
Badges (XML)	32.7 MB	2.5 MB (72s)	2.9 MB (69s)	2.8 MB (65s)	**1.9 MB (13s)**
Users (XML)	48.0 MB	7.6 MB (52s)	8.1 MB (51s)	8.1 MB (89s)	**6.4 MB (22s)**
Gandhi Works	100.6 MB	20.7 MB (**90s**)	20.9 MB (95s)	22.5 MB (213s)	**20.3 MB** (92s)
GloVe Emb.	193.4 MB	57.9 MB (261s)	57.9 MB (**239s**)	60.0 MB (426s)	**57.3 MB** (315s)
			Logs & SQL Datasets		
Sakila DB	8.7 MB	426 KB (23s)	501 KB (23s)	466 KB (23s)	**298 KB (2.6s)**
Weblog Sample	67.6 MB	2.7 MB (51s)	2.9 MB (78s)	3.1 MB (177s)	**2.5 MB (35s)**
Logfiles	242.0 MB	13.0 MB (203s)	13.3 MB (258s)	14.1 MB (572s)	**10.2 MB (99s)**
Audit Dump (SQL)	64.6 MB	12.0 MB (110s)	12.6 MB (106s)	12.1 MB (125s)	**10.1 MB (33s)**

4.2 Throughput Analysis (Rust Implementation)

A common drawback of pre-processing is added latency.
However, Table 2 shows that the Rust implementation of CAST (leveraging 7-Zip as a threaded backend) is faster than applying 7-Zip directly to the raw file.

Analysis: This speedup occurs because the columnar transformation reduces the entropy complexity for the backend encoder.
Although CAST adds a parsing step, the resulting data streams (columns of integers, columns of dates) are trivial for LZMA2 to compress, resulting in a net reduction of total CPU time.

Table 2: Performance Benchmark: Size & Time (Rust + 7-Zip, 8 Cores)

Dataset	**Original**	**LZMA2 (7-Zip)**	**CAST (Python)**	**CAST (Rust+7z)**	**Speedup**
CSV Datasets					
Balance Payments	33.1 MB	834 KB (2.02s)	**244 KB** (5.5s)	255 KB (**1.57s**)	**1.28x**
Migration Stats	29.2 MB	1.38 MB (4.64s)	**317 KB** (6.9s)	343 KB (**2.11s**)	**2.20x**
Subnational Tables	16.0 MB	824 KB (2.65s)	-	**344 KB (1.10s)**	**2.41x**
NZDep Life Tables	13.0 MB	1.20 MB (2.73s)	-	**883 KB (1.40s)**	**1.95x**
Custom 2020	207.9 MB	25.3 MB (89.4s)	**18.4 MB** (213s)	19.0 MB (**66.4s**)	**1.35x**
Custom 2018	668.3 MB	56.6 MB (**105s**)	-	**25.9 MB** (136s)	0.77x
IOT Temp	6.9 MB	787 KB (1.45s)	-	**724 KB (1.20s)**	**1.21x**
Sitemap Apple	124.2 MB	2.69 MB (**9.25s**)	-	**1.99 MB** (12.5s)	0.74x
Nashville Housing	9.9 MB	1.42 MB (2.36s)	-	**1.28 MB (2.05s)**	**1.15x**
Item Aliases	201.5 MB	40.6 MB (**83.7s**)	-	**40.2 MB** (97.0s)	0.86x
IoT Intrusion	197.5 MB	28.2 MB (99.7s)	-	**24.2 MB (74.4s)**	**1.34x**
LinkedIn Profiles	52.5 MB	4.57 MB (12.0s)	-	**4.03 MB (10.7s)**	**1.11x**
Gafgyt Botnet	105.8 MB	26.3 MB (74.9s)	**22.6 MB** (128s)	25.3 MB (**69.0s**)	**1.08x**
HomeC	131.0 MB	15.4 MB (54.6s)	**11.1 MB** (103s)	11.7 MB (**41.3s**)	**1.32x**
DDoS Data	616.8 MB	20.4 MB (81.1s)	**10.2 MB** (463s)	10.9 MB (**71.9s**)	**1.13x**
Wireshark P3	154.4 MB	10.6 MB (47.7s)	**5.8 MB** (145s)	6.94 MB (**35.8s**)	**1.33x**
RT_IOT2022	54.8 MB	2.56 MB (**8.66s**)	**1.99 MB** (23.5s)	2.01 MB (9.54s)	0.91x
Metasploitable	52.8 MB	3.87 MB (**11.3s**)	**3.5 MB** (28.0s)	3.52 MB (11.8s)	0.96x
OWID Covid	46.7 MB	7.20 MB (15.7s)	**6.3 MB** (29.4s)	6.36 MB (**14.2s**)	**1.10x**
Assaults 2015	234 KB	**34.4 KB (0.06s)**	39.5 KB (0.18s)	39.9 KB (0.08s)	0.75x
JSON & XML Datasets					
Wikidata Fanout	262.3 MB	33.4 MB (139s)	-	**29.2 MB (124s)**	**1.12x**
Gandhi Works	100.6 MB	20.8 MB (55.4s)	**20.3 MB** (91.5s)	**20.3 MB (55.2s)**	1.00x
Badges (XML)	32.7 MB	2.56 MB (9.16s)	**1.9 MB** (12.8s)	1.95 MB (**4.06s**)	**2.25x**
Users (XML)	48.0 MB	7.71 MB (15.8s)	**6.4 MB** (21.9s)	6.43 MB (**9.57s**)	**1.65x**
Votes (XML)	145.8 MB	6.20 MB (30.8s)	**3.6 MB** (57s)	3.92 MB (**12.9s**)	**2.39x**
Yelp Business	118.9 MB	11.1 MB (32.5s)	-	**10.9 MB (26.1s)**	**1.24x**
Yelp Tips	180.6 MB	35.0 MB (79.3s)	-	**30.4 MB (58.6s)**	**1.35x**
Yelp Checkin	287.0 MB	55.0 MB (**157s**)	-	**54.2 MB** (167s)	0.94x
Parent-Child Dict	214.5 MB	29.5 MB (120s)	-	**28.8 MB (111s)**	**1.08x**
Train.json	11.9 MB	1.85 MB (3.06s)	-	**1.80 MB (3.03s)**	**1.01x**
Examples Train	201.4 MB	7.73 MB (39.3s)	-	**4.68 MB (26.1s)**	**1.51x**
Wiki Text 1	41.2 MB	**10.3 MB (17.7s)**	-	**10.2 MB** (19.1s)	0.93x
Wiki Text 2	41.5 MB	**10.1 MB (17.4s)**	-	**10.0 MB** (18.7s)	0.93x
Glove Emb.	193.4 MB	58.1 MB (**179s**)	**57.3 MB** (315s)	57.8 MB (195s)	0.92x
Pagerank	121.9 MB	15.8 MB (**45.4s**)	-	**15.7 MB (48.6s)**	0.94x
Brazil Geo	14.5 MB	**1.55 MB (2.52s)**	-	1.55 MB (3.03s)	0.83x
Logs & SQL Datasets					
Logfiles	242.0 MB	15.7 MB (52.6s)	**10.2 MB** (99s)	11.9 MB (**39.2s**)	**1.34x**
Weblog Sample	67.6 MB	3.16 MB (**9.09s**)	**2.5 MB** (34.6s)	2.90 MB (9.52s)	0.95x
Dynamic Audit	64.6 MB	12.4 MB (27.1s)	10.1 MB (32.8s)	**10.0 MB (16.0s)**	**1.69x**
Sakila Insert	8.8 MB	492 KB (**0.97s**)	298 KB (2.6s)	**297 KB** (1.04s)	0.93x
Xdados	4.4 MB	533 KB (**1.10s**)	-	**433 KB** (1.50s)	0.73x
PCAP Dump	0.9 MB	**144 KB (0.18s)**	-	144 KB (0.18s)	1.00x
IP Capture	38.7 MB	**18.3 MB (3.39s)**	-	18.3 MB (3.50s)	0.97x

5 Limitations & Constraints

1. **Binary & High Entropy Data:** As evidenced by benchmarks on PCAP (Network Dump) files, CAST provides no benefit for binary 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 (Python Reference and Rust Performance Preview) are designed as **scientific proofs-of-concept**.
   While functional, 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`) and a Python 3.10 reference implementation.
Exact invocation commands are documented in the repository. Additionally, the benchmarking CLI tool is designed to explicitly output the internal decision metrics (Unified/Split mode, parsing strategy, template counts) to the standard output during execution, allowing the adaptive behavior of CAST to be inspected and verified for each run.

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`