CAST: Columnar Agnostic Structural Transformation

A Schema-less Structural Preprocessing Algorithm for Enhancing Compression Density and Throughput on Structured and Semi-Structured Data Streams

Andrea Olivari
github.com/AndreaLVR

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 **structured and semi-structured data streams**, 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. **Contrary to the traditional trade-off between space and time, CAST reduces end-to-end compression time compared to general-purpose compressors such as LZMA2, Zstandard, and Brotli, when these baselines are configured for their maximum compression efficiency.** By normalizing structural entropy, the algorithm lowers the computational burden on the backend encoder, allowing for simultaneously higher compression ratios and faster throughput. On highly repetitive structured datasets, compression ratios significantly exceed those obtained by standalone compressors, demonstrating that lightweight, schema-free structural normalization can substantially enhance compression density and transmission efficiency without modifying the underlying encoding algorithms.

1 Introduction

Structured and semi-structured data (logs, IoT telemetry, database dumps, CSV reports) 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 optimized for **storage efficiency and bandwidth-limited transmission**: 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 **Reference Rust implementation**[1] (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 compression throughput, achieving substantial speedups with negligible compression loss compared to the native integration, as demonstrated in Table 2.

Recognizing these distinct performance characteristics, the reference implementation adopts a **hybrid default strategy**: it automatically utilizes the System backend (7-Zip) for **compression** to maximize multi-threaded throughput, while defaulting to the Native backend for **decompression** to minimize process-spawning latency and overhead.

Note: A conceptual non-optimized prototype in Python is also provided in the repository solely to illustrate the core logic via readable Regular Expressions; however, due to its interpretive overhead, it is excluded from the performance analysis.

2 Methodology

The fundamental premise of CAST leverages the established concept that structured text lines L can be decomposed into a static template S and a variable vector V (**a model shared with log-mining literature**):

$$L \to S + V$$

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 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).

[1] While this paper validates the *Reference Implementation* (Standard) to establish the algorithmic baseline, a **Work-In-Progress Early Research prototype** of the next-generation architecture (featuring Row Groups for $O(1)$ Random Access) has been released in the repository's `rust_random_access_PREVIEW` directory. Preliminary tests confirm the feasibility of instant $O(1)$ retrieval with minimal compression overhead; however, **full-file decompression is currently unoptimized** pending the integration of the Standard version's buffered I/O strategies.

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. To ensure scalability on datasets exceeding physical memory, CAST operates on a streaming block-based architecture, as formally described in Algorithm 1.

Algorithm 1 CAST Streaming Compression Logic

Input: Byte stream D, Chunk size S (default: $S \geq |D|$)
Output: Sequence of compressed blocks B_{out}
while D has data **do** ▷ Streaming loop (main driver)
    $C \leftarrow$ read next S bytes from D
// Step 1: Binary Guard (per-chunk)
    if IsLikelyBinary(C) **then**
        Emit BackendCompress(C) with flag 255
        continue
    end if
// Step 2: Heuristic Analysis (per-chunk)
    $Strategy \leftarrow$ **AnalyzeSample**(C) ▷ Strict vs Aggressive
    $Map \leftarrow \{\}$, $Skeletons \leftarrow []$, $Columns \leftarrow \{\}$, $StreamIDs \leftarrow []$
// Step 3: Decomposition (Transformation)
    for line in C **do**
        if ContainsReservedChars(line) **then** ▷ Collision Fail-Safe
            Emit BackendCompress(C) with flag 255; **break**
        end if
        $S, V \leftarrow$ **Mask**($line, Strategy$)
        if $S \notin Map$ **then**
            if templates_exceed_budget() **then** ▷ Entropy Fail-Safe
                Emit BackendCompress(C) with flag 255; **break**
            end if
            $Map[S] \leftarrow$ **NewID**(), $Skeletons$.**append**(S)
        end if
        $Columns[Map[S]]$.**append**(V), $StreamIDs$.**append**($Map[S]$)
    end for
// Step 4: Finalization (Unified vs Split)
    $Decision \leftarrow$ **DecideUnifiedOrSplit**($Columns, Skeletons$)
    if $Decision$ is **Unified then**
        $Payload \leftarrow$ **Pack**($Skeletons, StreamIDs, Columns$)
        Emit BackendCompress($Payload$) ▷ e.g., LZMA2, Zstd
    else
        Emit BackendCompress($Skeletons, StreamIDs, Columns$) separately
    end if
end while

Algorithm 2 CAST Streaming Reconstruction Logic

Input: Compressed sequence B_{in}
Output: Reconstructed byte stream D_{out}
while B_{in} has data **do** ▷ Process stream chunk-by-chunk
    Read Block Header (Lengths, CRC, Flag)
    if Flag is 255 (Passthrough) **then**
        $C \leftarrow$ Read compressed payload
        Emit BackendDecompress(C)
        continue
    end if
    // Step 1: Backend Inflation
    $R, I, V \leftarrow$ **BackendDecompress**(Payload components)
    $Skeletons \leftarrow$ **ParseRegistry**(R)
    $StreamIDs \leftarrow$ **ParseIDs**(I)
    // Step 2: Queue Population ($O(N)$)
    $Queues \leftarrow$ Initialize variable queues per column from V
    // Step 3: Row Reconstruction
    for id in $StreamIDs$ **do**
        $S \leftarrow Skeletons[id]$
        $Row \leftarrow$ Empty String
        for segment in S **do**
            if segment is Placeholder **then**
                $Var \leftarrow Queues[id]$.**pop_front**()
                Row.**append**(**Unescape**(Var))
            else
                Row.**append**($segment$)
            end if
        end for
        Emit Row
    end for
end while

Streaming and Scalability By default, CAST operates in **Solid Mode**, treating the entire input as a single chunk to maximize global redundancy extraction. However, the architecture natively supports **Streaming Chunking** via the parameter S. **It is important to note that the decompression memory footprint is dictated by the encoding strategy:** while the architecture allows for stream-based processing, processing a file compressed in Solid Mode requires buffering resources proportional to the compressed block size. Therefore, O(S) memory complexity is guaranteed only when the input has been encoded with explicit chunking enabled; otherwise, the decompressor adapts to the block size defined at compression time.

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, LogHub, Zenodo, HuggingFace** 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 a **unified high-performance Rust engine** capable of operating in two distinct modes. **While integrated into a single artifact, these modes are architecturally decoupled to prevent benchmark pollution:** this internal separation ensures that the algorithmic efficiency metrics are derived purely from the transformation logic (Native), strictly isolated from the OS-level piping and binary invocation overhead inherent to the production pipeline (System).
Both configurations feature native `multi-threading` capabilities and configurable memory parameters (specifically `chunk-size` and `dict-size`) to strictly control the runtime footprint (**via block segmentation in compression and streaming I/O in decompression**), preventing memory saturation and enabling the processing of datasets larger than available RAM.
We evaluated this engine in two specific configurations:

- **Native Mode (Reference Configuration):** Selected via the `--mode native` flag, it links directly against native libraries.
  While this configuration 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 (High-Throughput Configuration):** Selected via the `--mode 7zip` flag, it 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.

Backend Flexibility & Tuning. Crucially, since CAST operates as a transparent preprocessor, the end-to-end performance characteristics are tunable via backend selection.
A user prioritizing restoration speed could pair CAST with a real-time compressor (e.g., Zstd at moderate levels), effectively shifting the trade-off towards "warm" storage requirements. Conversely, for deep archival scenarios where storage density is paramount, a high-ratio backend is preferred. To demonstrate the maximum potential for storage minimization (the primary focus of this work), the following benchmarks evaluate the high-compression **CAST + LZMA2** pipeline.

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.

The "Extreme Density" Baseline (ZPAQ). While the primary comparison focuses on production-ready algorithms, we also cross-referenced results against ZPAQ (v7.15, -m5). ZPAQ is widely recognized as a benchmark for extreme archival compression, typically offering superior density at the cost of impractical runtimes (often orders of magnitude slower than LZMA2). Consequently, it is not included in the standard charts; however, specific instances where CAST

outperforms even this extreme density baseline are explicitly highlighted in **Table 1** with a red asterisk (*).

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 Native Mode Benchmarks

Table 1 details the reduction in file size and processing time across the full dataset corpus using the Rust Native implementation. To visually contextualize these findings without overcrowding the charts, we refer the reader to **Figure 1** for a ranked comparison of algorithmic efficiency on a *representative selection* of these datasets, and **Figure 2** for the corresponding throughput analysis.

Table 1: Native Mode Benchmarks: Algorithmic Efficiency Comparison (Single Thread)

Dataset	Original	LZMA2	Zstd	Brotli	CAST (Native)	Ratio
CSV Datasets & Batch scripts						
Caltech Kepler Q4 Wget (bat)	63.1 MB	572 KB (349s)	1.36 MB (315s)	1.35 MB (211s)	**198 KB (19.7s)**	**319x** *
Balance Payments	33.2 MB	501 KB (110s)	698 KB (103s)	592 KB (131s)	**245 KB (4.7s)**	**136x** *
Caltech KELT S14 Wget (bat)	126.9 MB	2.40 MB (115s)	3.63 MB (96s)	3.79 MB (440s)	**1.10 MB (18.1s)**	**115x** *
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**
Logs and SQL Datasets						
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 (165s)**	**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** *
JSON & XML Datasets						
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**
Text Datasets						
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**

* *Denotes datasets where CAST outperforms even ZPAQ (m5) in compression density (and typically speed).*
CAST vs ZPAQ: Kepler (198 KB vs 230 KB); KELT (1.10 MB vs 1.33 MB); Migration (319 KB vs 471 KB); Balance (245 KB vs 250 KB); Spider NLP (406 KB vs 461 KB); RT_IOT (1.99 MB vs 2.00 MB); Wireshark (5.69 MB vs 6.95 MB); HAI Security (13.0 MB vs 13.1 MB); US Stock (17.4 MB vs 17.6 MB); Pixel 4XL (25.0 MB vs 27.3 MB).

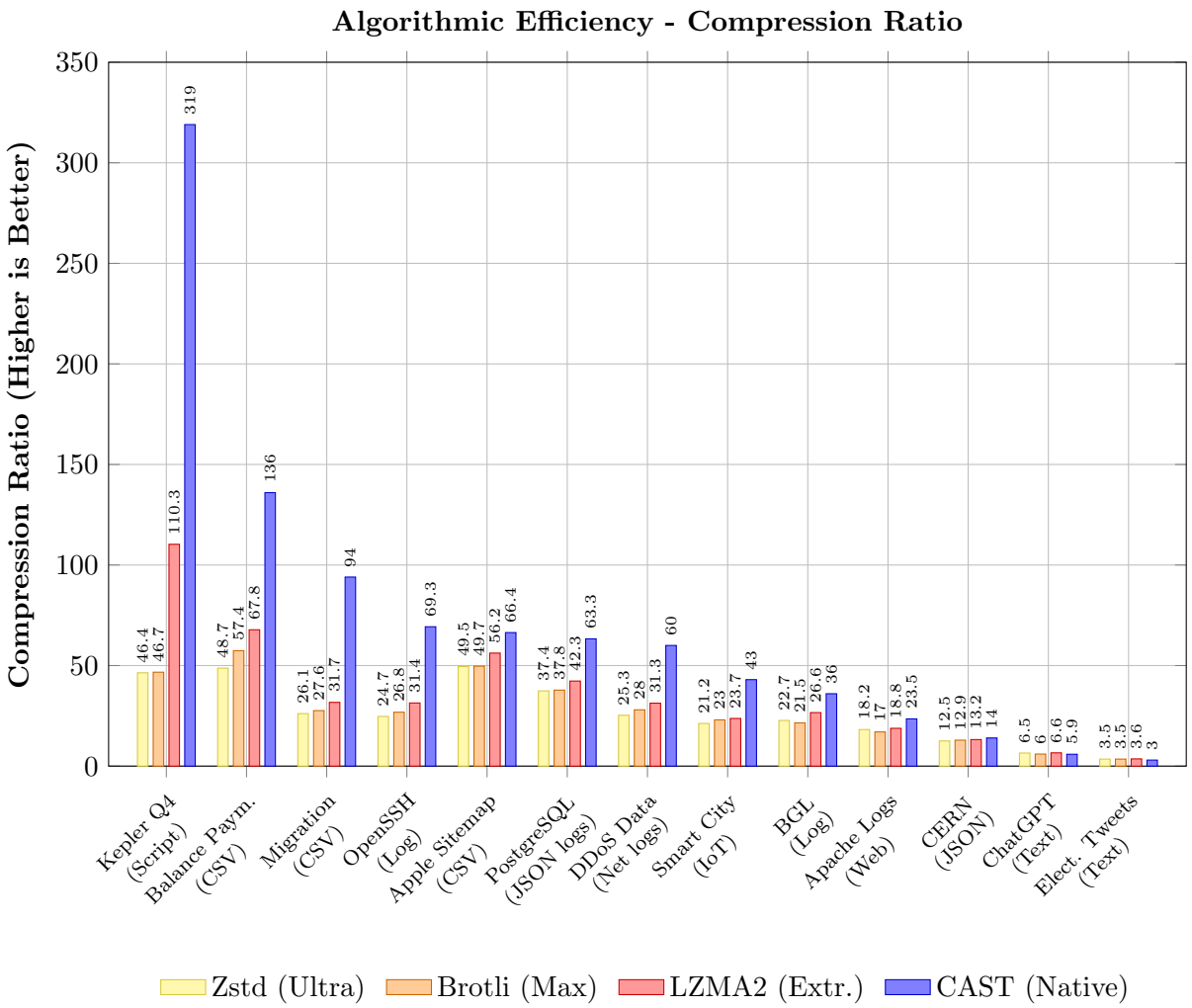

Figure 1: **Algorithmic Efficiency Gap:** Comparison of compression ratios across 13 datasets. The chart visually demonstrates the massive efficiency gap on structured data: **Kepler Q4** (first column) reaches a **319x** ratio with CAST, dwarfing traditional algorithms (LZMA2 110x). This trend of 2x-3x improvement persists across CSV, Logs, and IoT data.

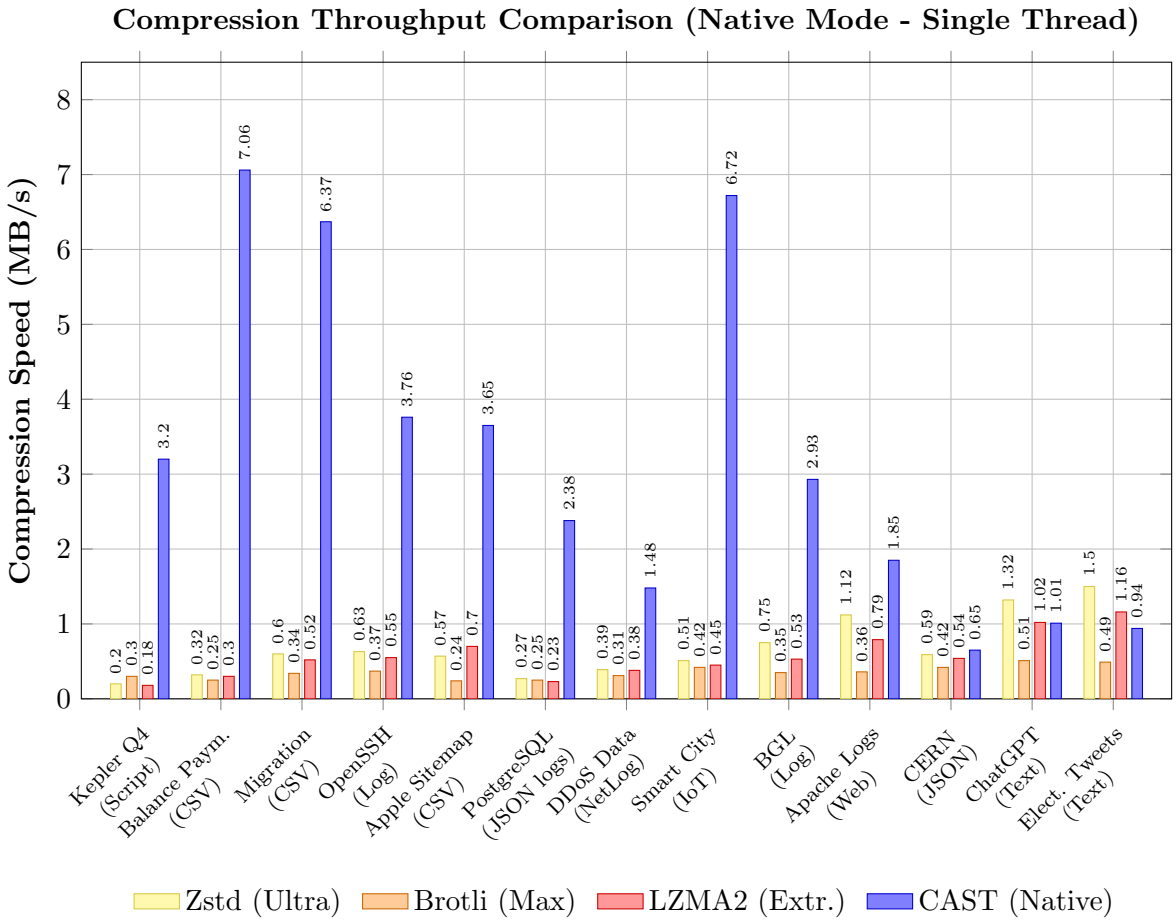

Figure 2: **Throughput Analysis (Native Mode):** Comparison of compression speeds (MB/s). CAST (Blue) demonstrates superior throughput on structured data (e.g., *Balance Payments* at 7.06 MB/s, *Smart City* at 6.72 MB/s) by significantly reducing the data volume before it reaches the backend compressor. On purely unstructured text (right side), speed converges with standard algorithms.

4.2 System Mode 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.

Crucially, the benchmarks honestly report the performance regression on unstructured text (e.g., ChatGPT, Tweets), where the parsing overhead is not offset by compression gains. This confirms the hypothesis that CAST's advantage is strictly structural.

To visualize this phenomenon, **Figure 3** correlates the processing throughput with the compression ratio. The data reveals a clear trend: as the structural density increases (higher ratio), the backend encoding speed accelerates proportionally, confirming that CAST acts as a throughput multiplier for structured data.

Table 2: System Mode Benchmarks

Dataset	Original	LZMA2 (7-Zip)	CAST (Native)	CAST (System)	Compression Ratio	Speedup (vs LZMA2)
CSV Datasets						
Caltech Kepler Q4 Wget (bat)	63.1 MB	1.18 MB (10.5s)	**198 KB** (19.7s)	206 KB (**3.3s**)	**307x**	**3.21x**
Balance Payments	33.1 MB	834 KB (2.02s)	**245 KB** (4.7s)	255 KB (**0.75s**)	**129x**	**2.69x**
Caltech KELT S14 Wget (bat)	126.9 MB	3.02 MB (20.2s)	**1.10 MB** (18.1s)	1.12 MB (**5.4s**)	**114x**	**3.74x**
Migration Stats	29.3 MB	1.38 MB (4.64s)	**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**)	**41.9x**	**5.88x**
COVID-19 Surveillance	872.1 MB	32.8 MB (63s)	**20.3 MB** (559s)	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)	**5.9x**	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**
Logs & SQL Datasets						
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**)	**35.6x**	**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**
JSON Datasets						
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
Text Datasets						
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.*

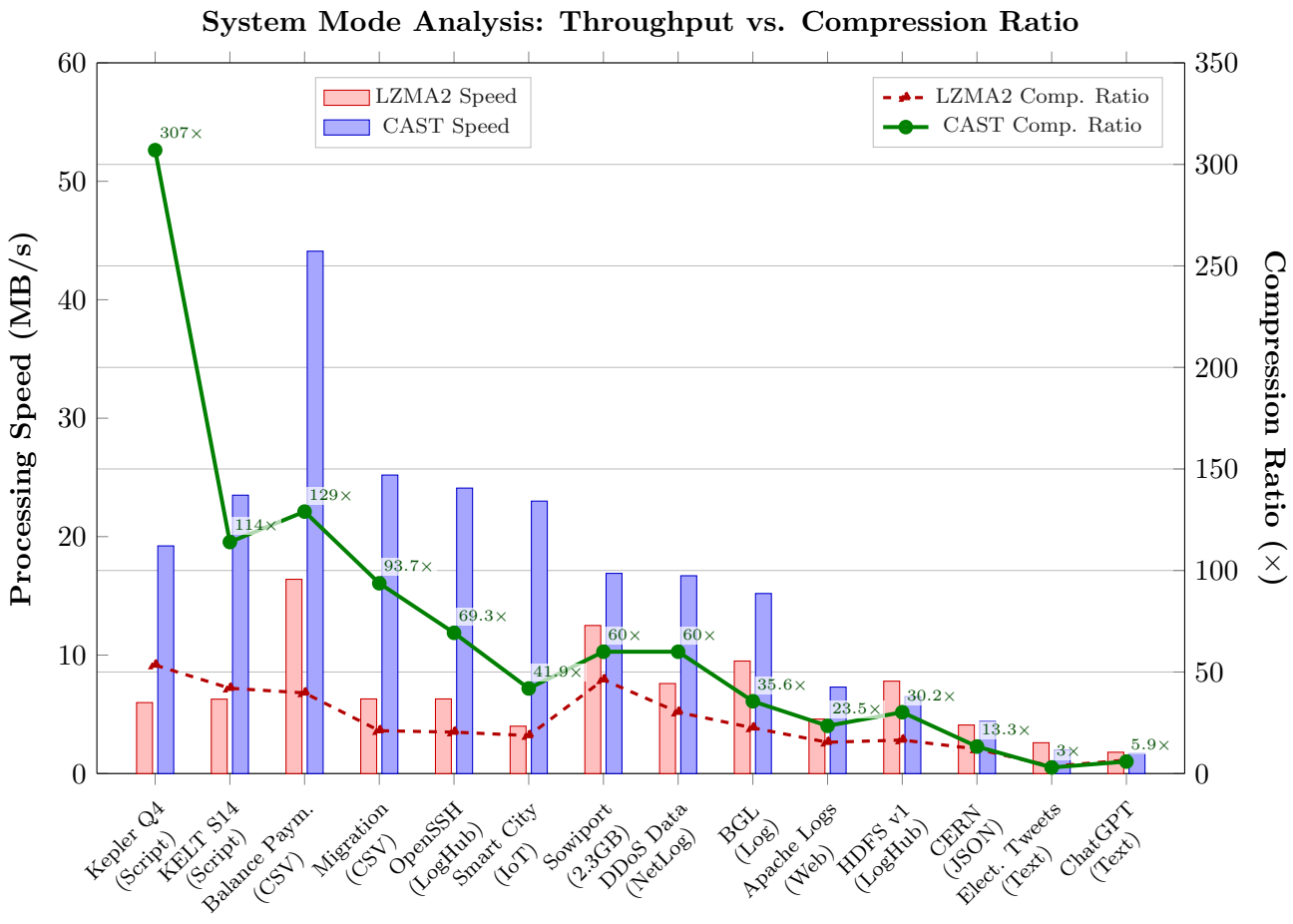

Figure 3: **System Mode Analysis.** The chart reveals a strong positive correlation between compression efficiency (Green Line) and processing throughput (Blue Bars). The new entries (*Kepler*, *KELT*) confirm this trend, showing massive ratios (up to 307x) and high throughput (around 20 MB/s) in a multithreaded environment.

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.

- **The Cost of Reconstruction:** As evidenced in **Figure 4**, the raw LZMA2 decoder generally outperforms the CAST pipeline. This is an expected consequence of the architectural design: while LZMA2 simply inflates a byte stream, CAST must perform memory-intensive structural reassembly. On complex CSVs (e.g., *NYC Bus*) or high-entropy logs (e.g., *HDFS*), this overhead results in a 20-50% reduction in throughput compared to the baseline.
- **Operational Viability:** Despite the comparative slowdown, CAST consistently delivers decompression speeds between **50 MB/s and 200 MB/s**. While slower than the raw backend, this performance remains competitive with standard HDD read speeds and is sufficient for batch processing and archival restoration workflows. Notably, on highly

repetitive datasets (e.g., *Smart City*, *US Stocks*), the reduced volume of data to decompress allows CAST to actually **outperform** the raw LZMA2 baseline, proving that under optimal conditions, the reduced I/O load can offset the CPU reconstruction cost.

- **Native vs. External Backend:** Benchmarks indicate that the **Native implementation is superior for the majority of datasets**. By eliminating the Operating System overhead associated with spawning external processes and piping data streams, the Native backend offers lower latency and consistent performance. The System Mode (7-Zip) retains a slight advantage only on specific high-entropy or extremely large workloads where the external decoder's internal threading model outweighs the IPC (Inter-Process Communication) cost. Consequently, CAST defaults to the Native engine for restoration tasks.

Table 3: Decompression Time Comparison: CAST (Native vs System) vs LZMA2

Dataset	Original Size	CAST (Native)	CAST (System)	LZMA2 (7-Zip)	CAST Speed (MB/s)	LZMA2 Speed (MB/s)
Smart City Sensors	23.04 MB	**0.13s**	0.15s	0.42s	**177.2**	54.8
HDFS v1 (LogHub)	1.47 GB	7.63s	8.78s	**6.21s**	197.3	**242.4**
BGL (LogHub)	708.8 MB	4.46s	4.40s	**2.1s**	161.1	**337.5**
Recipes JSON	85.2 MB	3.73s	4.34s	**3.1s**	22.8	**27.5**
PostgreSQL JSON Logs (Zenodo)	392.8 MB	2.39s	2.45s	**1.35s**	164.3	**290.9**
US Stock Prices	213.82 MB	**2.04s**	2.08s	2.3s	**104.8**	92.9
CERN OpenData Slim	202.12 MB	2.00s	2.01s	**1.87s**	101.1	**108.1**
IOTA logs 2.21 (merge)	1.48 GB	11.62s	18.26s	**5.4s**	130.4	**280.6**
HAI Security Train	108.91 MB	1.57s	1.40s	**1.32s**	77.8	**82.5**
ChatGPT Paraphrases	252.6 MB	3.16s	3.29s	**2.27s**	79.9	**111.3**
Pixel 4XL GNSS Log	124.43 MB	2.05s	**1.92s**	2.14s	**64.8**	58.1
PaySim Mobile Money	470.6 MB	9.07s	9.67s	**4.61s**	51.9	**102.1**
NYC Bus Breakdowns	126.71 MB	1.73s	1.82s	**1.02s**	73.2	**124.2**

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 buffered streaming and direct memory copying of pre-calculated offsets. Consequently, the structural restoration imposes **low computational overhead**, ensuring that the decompression process remains efficient and scalable even on large datasets.

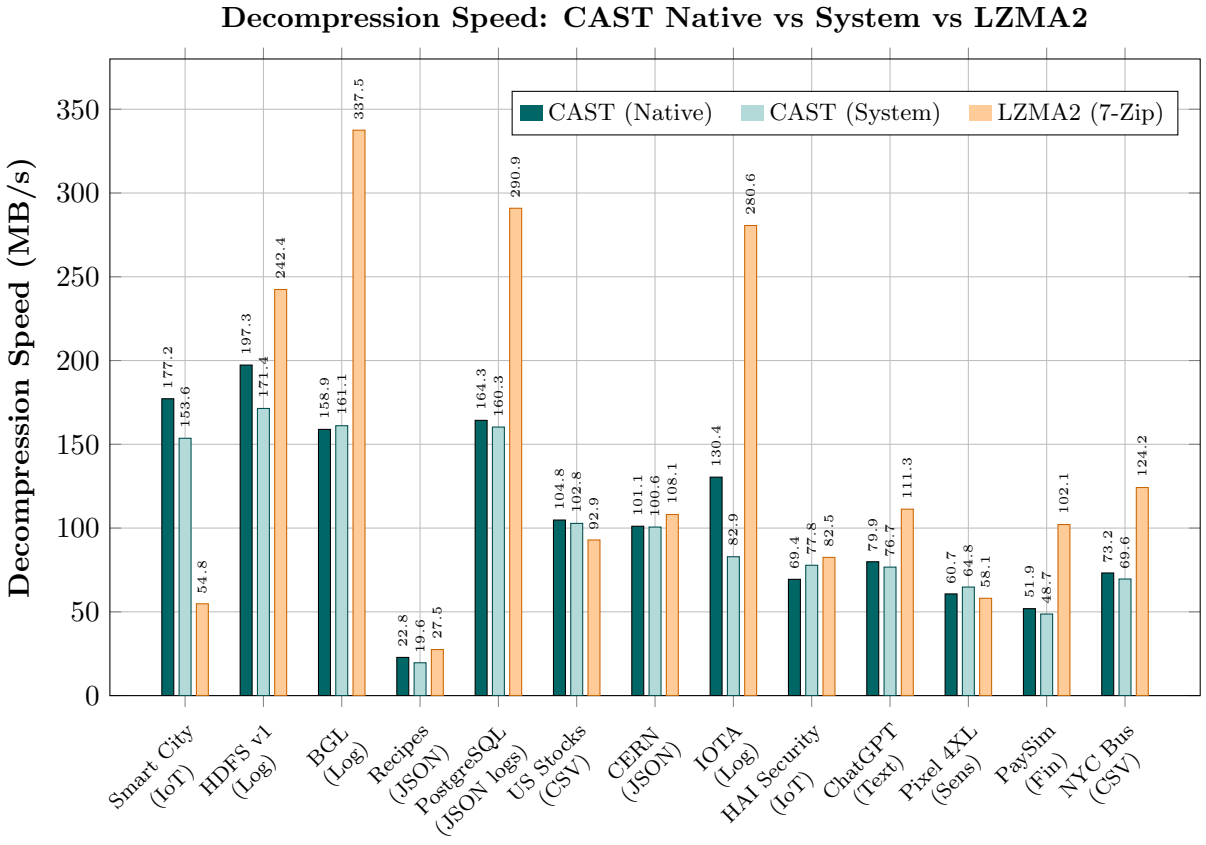

Figure 4: **Decompression Performance Landscape.** The chart benchmarks the native Rust implementation of CAST against the system-call variant and standard LZMA2. While the native approach (dark teal) effectively reduces runtime overhead compared to the system-based version (light teal), **LZMA2 (orange) maintains a higher throughput on the majority of datasets.** This performance gap quantifies the **trade-off between storage density and restoration speed**: CAST invests additional CPU cycles in the structural reconstruction phase ($O(N)$) to achieve higher compression ratios. Despite this overhead, CAST maintains operational viability with speeds ranging from 50 to 200 MB/s, validating its use case for archival storage where density is prioritized over instant retrieval.

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. **Maturity & Performance Profile:** Significant engineering effort has been dedicated to ensuring the robustness and stability of the provided implementations, incorporating strict memory safeguards and error handling strategies. However, as a recently developed research prototype, it naturally lacks the extensive, multi-year validation and security

auditing characteristic of legacy standard utilities like `xz` or `zstd`. Nevertheless, empirical testing suggests a high degree of reliability and efficiency; in particular, the **System Mode** pipeline demonstrates exceptional potential, delivering throughput that frequently exceeds the standalone 7-Zip baseline, validating the practical viability of the approach. Future development is committed to addressing emerging edge cases and further hardening the codebase, ensuring a continuous evolution towards full production readiness.

5. **Backend Agnosticism & Scalability:** CAST functions as a generic pre-processor compatible with any stream compressor. This architecture allows users to optimize the pipeline: pairing CAST with high-throughput backends (e.g., Zstd) **directly improves end-to-end restoration speed** by minimizing the raw decompression phase. While the structural reconstruction step imposes a fixed computational baseline, the overall system scales effectively with the backend's performance, allowing users to select the optimal trade-off between compression ratio (LZMA2) and access speed (Zstd).

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 Formats such as Apache Parquet and ORC utilize columnar layout to optimize analytic workloads (OLAP). While efficient, they typically rely on **schema-on-write** paradigms, requiring explicit definition or costly inference steps upfront. They are widely adopted for structured data lakes rather than raw log archival. CAST aims to extend entropy reduction benefits to schema-less, semi-structured streams without enforcing strict typing or mandating a rigid ETL process.

Log-specific Compressors State-of-the-art systems like CLP, Crystal, and OpenZL operate on decomposition principles similar to CAST: they separate static templates from dynamic variables to model semi-structured data. However, a primary distinction lies in the architectural scope. Tools like CLP are often engineered as **holistic storage and query engines**, employing specialized encodings designed to support direct searchability and granular random access on compressed data.
In contrast, CAST is designed as a modular *transformation layer* for *deep archival*. Instead of introducing a custom query-able format, it reshapes data to maximize the efficacy of general-purpose, high-ratio compressors (like LZMA2). By decoupling the structural transformation from the encoding backend, CAST prioritizes maximal storage density and portability over immediate query execution capabilities. Nevertheless, the underlying block-based architecture of CAST shares the potential for future indexed extraction, bridging the gap between pure archival formats and query-ready systems (a preliminary prototype is discussed in Section 8).

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. Consequently, CAST acts as a macroscopic pre-processor that is largely orthogonal and complementary to these bit-level or symbol-level 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 partic-

ularly 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 & Experimental Preview. To represent a concrete step towards query-ready storage, we have released an **experimental Early Research evolution** of the engine, available in the repository's `rust_random_access_PREVIEW` directory (and as a pre-compiled Preview release). This preview implements **Independent Row Groups** and **Footer Indexing** to enable $O(1)$ random access. While the implementation is extremely recent and validated on a non-exhaustive dataset, preliminary benchmarks suggest the viability of this architecture. Initial tests show near-instantaneous retrieval times with a marginal compression trade-off (typically $+5\% \sim 10\%$) compared to the Standard version. **While current decompression throughput is promising, we anticipate further performance gains once the dedicated CPU optimizations and zero-copy memory strategies of the Standard Native implementation are fully integrated into this experimental prototype.**

Availability

The complete source code, including the reference implementation and experimental artifacts, is available under the MIT License at:
`https://github.com/AndreaLVR/CAST`