

DictForge: Measuring and Improving Zstandard Dictionary Training

Yugen
TODO: affiliation
19thkingisreal@gmail.com

Abstract

Zstandard’s dictionary trainer (COVER/fastCOVER) [15] is the de facto standard for small-record compression. However, it has received no algorithmic attention since 2018: requesting a larger dictionary can silently yield a smaller one, and construction ignores size and level. Existing tools reuse the same heuristic and a fixed objective that cannot tell an efficient dictionary from a starved one. This paper presents a validation-driven, level-aware trainer that searches size and trainer family under an anytime never-worse guarantee, then refines content and seeds offsets at high levels on five corpora. Against the deployed **zstd -train** default, our trainer improves compression ratio by 3.1–14.3% at level 3 and 11.5–32.1% at level 19. Matched-size rebuilds show up to 6.9% of the level-19 gain survives at identical budgets, versus 0–11% at level 3.

1 Introduction

Dictionaries are how zstd compresses small data, and small data is where modern systems live: key-value stores, logs, telemetry, and the emerging shared-dictionary web transport [17]. One trainer serves all of it: COVER and its fast approximation fastCOVER [15], a 2016 heuristic frozen since 2018, whose own maintainer has stated that the ideal dictionary builder “does not exist.”

Yet the trainer has documented pathological failures. A production engineer reported that raising **-maxdict** from 512 KB to 550 KB collapsed compression ratio from 90× to 14× on their data [14]; the thread was closed without an explanation, and the trainer’s behavior is unchanged in the current release. We show this signature is reachable on zstd’s own benchmark corpus and trace it to a specific selection-objective defect, alongside two further defects: the trainer never searches dictionary size, and it optimizes construction decisions for a single compression level regardless of which level the caller actually uses.

Guided by this diagnosis, we build a trainer that treats dictionary construction as a search adjudicated by measurement rather than by proxy, with every stage gated on held-out validation rather than on the training data it searched over.

Our contributions are as follows. We give the first mechanistic, quantitative diagnosis of the incumbent trainer, tracing budget degeneracy to its selection objective, quantifying size blindness, and confirming level blindness three independent ways; the budget-degeneracy finding is consistent with a sensitivity collapse reported upstream but never explained [14]. We build a level-aware, validation-driven trainer with an anytime-safety guarantee (Propositions 1 and 2),

each stage individually gated and validation-checked. We evaluate it on a 5-corpus, 2-level benchmark against the deployed default, a klauspost `builddict` [18] baseline, an equal-wall-clock stock sweep, a no-training control, and matched-size rebuilds at Cassandra’s, RocksDB’s, and (a setting motivated by) ScyllaDB’s production budgets, isolating how much of the headline gain is size versus method. The resulting tool emits 100%-standard-format dictionaries, requiring no change to any zstd library or client, and is open source together with all experiments.

2 Related Work

Dictionary construction. The trainer descends from Liao, Petri, Moffat, and Wirth’s relative Lempel-Ziv dictionary construction [15], a covering problem over sampled substrings; zstd adopted it as `-train-cover` and later added the fastCOVER approximation that is today’s CLI default, unchanged since 2018 and never described in a paper of its own. We reuse the covering heuristic unmodified; our contribution is the selection layer around it. Level-dependence is not a new observation – a zstd maintainer noted in 2019 that training at the level you actually compress at “will tune the dictionary better” [16], and a 2022 request to expose a level parameter was declined for ABI stability [3] – but the *mechanism* (three independent, opposite-sign inversions traced to hash-table overwrite vs. full indexing, §4.3) and *magnitude*, including the first measurement of the disabled reopcode-seeding path, are new. `klauspost/compress` [18] is the only other tool we know of that emits standard-format zstd dictionaries and exposes a level-targeting flag; we benchmark it directly and it loses to the stock default on every cell we ran.

RLZ, methodology, and theory. Our refinement stage (§5.2) adapts RLZ reference pruning and selection [13, 4]: the novelty is doing it against the production encoder’s own sequence stream, at match granularity, gated by measured compressed size. Optimal dictionary choice is intractable in general – NP-complete [20], APX-hard as smallest-grammar [7, 6] – so our propositions (§5.3) deliberately constrain only the *selection* rule, not the underlying trainer; domain-specific compression [19] and LZ77 acceleration [21] are recent DCC neighbors in a different corner of the space. Methodologically we follow recent practical compression work that measures precisely why the incumbent underperforms before designing against it [5, 1], reporting losses (§7.2–§7.3) as well as wins.

Deployment. Meta’s managed-compression practice, disclosed only in a blog post [12], and the successor project OpenZL [10], treat per-workload specialization as known industrial practice with little published methodology; we target the specific, publicly reachable trainer instead. Deployment of trained dictionaries varies sharply by system and path (§3.2). The web’s shared-dictionary transport [17] and Brotli’s own dictionary-aware format [2] see modest uptake so far – Chrome usage sits at roughly 0.09% of page loads [11], and Cloudflare’s implementation remains a passthrough beta that itself generates no dictionaries [8] – context for why §7.2 matches specific deployment configurations rather than only our own chosen sizes.

3 Background

3.1 Dictionaries, indexing, and the trainer

A zstd dictionary is a self-describing binary artifact – magic number, ID, entropy tables (12 bytes of seeded recent-offset values), then `Content`, a virtual “past” matches may reference by offset [9]. Both loaders parse this independently and symmetrically, so a dictionary from any compliant writer is a drop-in replacement for `zstd -train`: *fact 1*. `Content` is not indexed uniformly, though: fast strategies hash it into one or two tables where later positions can overwrite earlier

ones, while high-level binary-tree strategies retain every position – how much of “the dictionary” is reachable is a function of level: *fact 2*, the mechanism behind level blindness (§4.3).

The trainer solves a covering problem: COVER [15] builds an exact suffix array over d -length substrings and scores each by document frequency, deduplicated per sample (a dictionary only helps the *first* reference to a substring); fastCOVER trades this for a lossy, streaming hash-table count. Both partition the corpus into epochs, greedily pack the best-scoring k -byte window per epoch into the output buffer back forward (strongest content nearest the data, cheapest to reference), and grid-search (d, k) , keeping whichever candidate achieves the smallest *measured* check-set compressed size – selection is literal compression, not a proxy. `zstd -train` with no flags, and `ZDICT_trainFromBuffer` used by every downstream integration we examine, both hard-code fastCOVER-optimize at $d=8$, with a 4-step grid on k , a 0.75 train/check split, scored at level 3, `-maxdict` defaulting to 110 KB; repeat-offset seeding is computed and discarded behind `#if 0`, and the size-response `shrinkDict` path is hard-coded off. All claims here were checked at file:line precision against `facebook/zstd main@82d322c` (v1.6.0).

3.2 Deployment reality

Production systems mostly call the same entry point, or, with no trainer, `ZDICT_finalizeDictionary`, which skips content selection and attaches only the format header around caller-supplied content. **RocksDB**’s dictionary path is opt-in and off by default; once enabled, it uses our baseline trainer path by default, at 110 KB, level 3. **ScyllaDB** trains RPC and SSTable dictionaries through the same 110 KiB call, but only one is deployed by default: RPC training is off, so our level-1 cell is a setting *motivated by* that path, not a production claim; SSTable dictionary compression, by contrast, has been that vendor’s default compressor since release 2025.4, so the 110 KiB/level-3 configuration we test as “the deployed default” is production-representative there. **Cassandra**’s CEP-54 is not merged or shipped – its tracking epic is still in progress – so we keep its 64 KiB code default as an evaluation target, not a production claim. Each budget used in §7.2 is labeled by what it is: shipped default, evaluation setting motivated by an off-by-default path, or unreleased branch.

4 Diagnosis (measurements on real corpora)

4.1 Budget degeneracy (the “cliff”)

Consider a user who has trained a dictionary with `zstd -train-cover` and raises `-maxdict` for a better one. On `github_users` – `zstd`’s own published benchmark sample set – this produces Figure 1’s step function: at requests of 262–524 KiB the trainer emits exactly what was asked and ratio holds near 10; past 1 MiB it emits only ~415–425 KiB regardless of how much more is requested, and ratio falls to 7.49 (a 24% loss) and then 6.95 (31%) – *smaller output at a larger request*, with no warning at any verbosity level. In August 2024 an engineer integrating `zstd` dictionaries at Roblox reported the same signature upstream – a 512 KB budget yielding a 90× ratio against 550 KB yielding 14× on their data [14] – in a thread closed without the trainer’s behavior changing. We do not reproduce that magnitude, but the mechanism below accounts for the signature.

Mechanism. Three interacting behaviors, traced to source and confirmed by instrumented rebuild against `facebook/zstd main@82d322c`, produce the collapse. **M1:** epoch count scales with the *requested* budget, not the corpus (`cover.c:734-749`) – at $k=50$ and a 2 MiB request, a 36 MB training set becomes ~42,000 sub-kilobyte epochs. **M2:** the build loop exits after a hard-

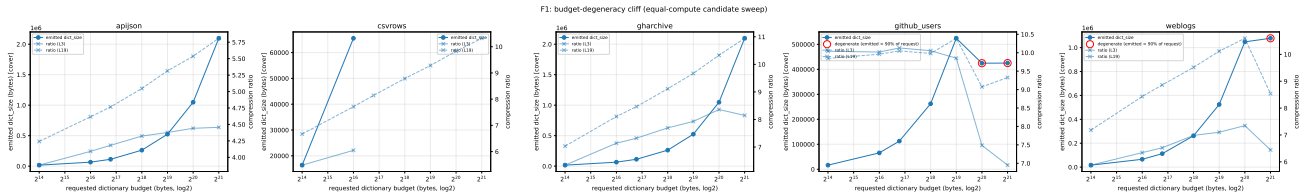


Figure 1: Requested dictionary size vs. emitted size and vs. compression ratio, one panel per corpus, marking degenerate builds (F1, generated by `tools/make_figures.py`).

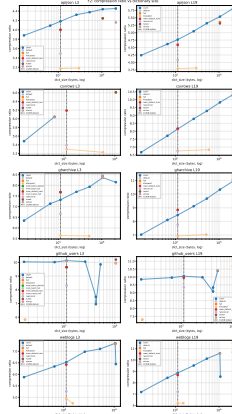


Figure 2: Compression ratio vs. dictionary size, one panel per (corpus, level), with the 110KB default marked (F2, generated by `tools/make_figures.py`).

coded run of dead epochs (`fastCOVER`: 10, `fastcover.c`:406), trivially reached on redundant corpora, leaving the output buffer substantially unfilled. **M3 (root cause)**: the selection objective scores each candidate by its *emitted* size plus check-set compressed size (`cover.c`:899,998), unable to distinguish “small because efficient” from “small because starved”; past a threshold the optimizer deterministically selects the most degenerate build – confirmed on `gharchive` at 2 MiB, where the winner had the *worst* compression term of all candidates and won purely on a 1.62 MB size advantage from its own failed fill. Roughly two thirds of the `github_users` collapse traces to this selection error, one third to genuine oversize dilution. A minor **M4** compounds it: finalize shrink-to-fit keeps the *first* bytes when trimming for the entropy header (`zdict.c`:902-935), discarding the tail – the highest-scoring content the builder placed nearest the data on purpose. Nobody noticed because both trainers assign `ctx->displayLevel` and then `memset` the context to zero a few lines later, silently suppressing every build-loop diagnostic at every `-v` level. M2, M3, and `displayLevel` are implementation defects; content-exhaustion pinning is inherent; `-maxdict` semantics are a documentation gap; we have drafted four upstream fixes.

4.2 Size blindness

Ratio-vs-budget curves from 2KB to 2MB (Figure 2) show peaks varying from 32KB to over 2MB depending on corpus and level; the default 110KB budget loses up to 22.4% (`gharchive`, level 19), and the curves are locally non-monotonic. The trainer searches (d, k) but never size; the `shrinkDict` path that would let it respond to size is dead code.

4.3 Level blindness (three independent confirmations)

The trainer optimizes for a single level, though construction decisions invert across levels. (a) Size-response inverts by level: levels 1–3 peak small then decline, while level 19’s response rises

monotonically. **(b)** Repcode seeding: +0.3–0.4% at level 19, –0.2–0.6% at level 1 – the first proper evaluation of the `#if 0` path. **(c)** Refinement: +1.7–6.6% at level 19, approximately 0 at level 3. Fast levels index only a suffix (hash overwrite) with greedy parsers, while `btopt` indexes everything and cost-models its choices; the same pattern should recur whenever the consumer is a different codec entirely – LZ4’s match window, for instance, is capped at 64 KiB – though we have not measured that cross-codec case in the rebuilt environment.

5 Method: a validation-driven, level-aware trainer

The diagnosis in §4 suggests a design rather than an algorithm: the stock trainer’s failures are failures of *selection*, not of its covering heuristic, which is sound and which we retain unchanged. We treat dictionary construction as blackbox optimization over a discrete candidate space – minimizing total compressed size of a corpus under a size-capped dictionary, an NP-hardness heritage traceable to Storer–Szymanski substitution [20] and smallest-grammar APX-hardness [7, 6] – with a ~ 1 -second exact oracle (a real compression run). Evaluation is cheap, so the design question is how to spend an evaluation budget.

Data discipline. The caller supplies one training directory, split 85/15 by seed into a *fit* set (what any underlying trainer sees) and a *validation* set (what every accept/reject decision is measured on); the user’s held-out data is never opened. This bounds overfitting in our greedy per-stage loops – the largest fit-to-heldout deviation observed is 0.14%.

5.1 Stage 1: size and family search

Stage 1 sweeps a geometric size ladder from 2 KiB to the caller’s cap with both fastCOVER and COVER, keeping whichever candidate scores best on validation at the target level. The *floor candidate* – fastCOVER at $\min(\text{cap}, 110 \text{ KiB})$, precisely what `zstd -train` would produce – runs first unconditionally, so every later decision is a comparison against what the user already had; this makes the tool never-worse by construction (Proposition 2). FastCOVER sizes run before COVER sizes (*cheap-first ordering*), so a truncated time budget cuts the expensive tail, not size coverage – a size-first ordering instead let large corpora exhaust the budget on COVER at small sizes, losing to the default by up to 19%. A *degeneracy guard* flags emitted size against requested size, but since selection is by measured compression, not requested size (§4.1, M3), a starved build cannot win by being small. A ladder winner is **refit on the full training directory** and kept only if validation does not regress – without this the tool loses $\sim 0.1\%$ to the default whenever no ladder entry beats the floor.

5.2 Stages 2 & 3 (target level ≥ 16): refinement and repcode seeding

Stage 1 chooses *which* dictionary; Stage 2 asks what the covering heuristic never asks: when real compressions run against it, which bytes do they reference? We extract the sequence stream from fit-set compressions, map matches to the regions they consumed, evict regions never touched or in the weakest referenced decile, and refill from the worst-compressing fit files, compacting proven content toward the tail (§4.1, M4). Each round is kept only if it improves fit-set compressed size, stopping at the first rejection, and the offset-to-region mapping self-tests against a planted synthetic region before the stage may run at all. Literally-dead content is rare (0–4%), so gains come from displacing *weakly* referenced content. Stage 3 seeds the three recent-offset values the encoder starts each frame with, which the stock trainer computes and discards, writing $\{1, 4, 8\}$: we measure the fit set’s top three offsets and patch them in, keeping the result only if validation improves. Both stages gate on target level ≥ 16 because fast levels index only part

of a large dictionary into small hash tables and parse greedily, so curated content and seeded offsets cannot be exploited and can actively mislead the parser, while level 19’s optimal parser indexes everything and cost-compares its options.

5.3 Guarantees

Let S be the size ladder, B the caller’s budget cap, and $C(B) = \{(t, s) : t \in \{\text{fastCOVER}, \text{COVER}\}, s \in S, s \leq B\} \cup \{\text{floor}(B)\}$ the Stage-1 candidate set, where $\text{floor}(B) = (\text{fastCOVER}, \min(B, 110 \text{ KiB}))$. The tool returns $\arg \max_{c \in C(B)} r_{\text{val}}(c)$, the candidate with the best measured validation-split compression ratio.

Proposition 1 (Budget monotonicity) *For $B_1 \leq B_2$, $C(B_1) \subseteq C(B_2)$, so the selected validation ratio is monotone non-decreasing in B .*

Proposition 2 (Anytime floor) *For any time budget sufficient to evaluate $\text{floor}(B)$ – seconds, and it is evaluated first – the selected validation ratio is at least that of the incumbent trainer at the same cap.*

Both are immediate from the construction; their content is not mathematical depth but the observation that the *deployed* trainer satisfies neither – Proposition 1 fails empirically by 24–31% (§4.1), and Proposition 2 fails because there is no floor at all, DCC-normal scope for a guarantee on a systems contribution. The guarantees hold on the validation measure, not held-out data (§7 quantifies the gap), and Proposition 1 constrains the *selected* candidate, not the underlying trainers – a starved build is still a starved build, we merely decline to select it.

6 Evaluation Methodology

Five small-record corpora spanning deployment shapes, including zstd’s own benchmark set: `github_users` (9,114 JSON user records), `gharchive` (20,000 GitHub events), `weblogs` (15,000 batches of 20 NASA-HTTP log lines), `apijson` (10,000 Crossref records), `csvrows` (10,000 50-row NYC 311 CSV fragments), each split 80/20 by seeded shuffle, with heldout data compressed exactly once per configuration. We evaluate at level 3 (RocksDB/ScyllaDB SSTable/Cassandra default) and 19 (archival), plus level 1 for a cell motivated by ScyllaDB’s off-by-default RPC path (§3.2), on an Apple M4 against zstd 82d322c (v1.6.0), every dictionary round-trip-verified against an independent stock binary; x86 replication of throughput is planned but not run.

We compare against five references: the deployed default (`zstd -train` at 110 KiB), a no-training control (random samples concatenated to the same budget), `klauspost/compress`’s `bulddict` [18], an equal-wall-clock sweep of the stock optimizer, and matched-size/production rebuilds at Cassandra’s 64 KiB budget, RocksDB’s opt-in no-trainer fallback, and the ScyllaDB-motivated level-1 setting (§3.2). Table 1 reports the *tuned* (Stage 1 only) and *full* (all stages) variants of our trainer against these baselines.

Midway through this work the machine’s temporary storage was cleared, destroying the experimental tree; we rebuilt it from recorded specifications – re-fetching every corpus, recompiling, re-running every experiment – and report only rebuilt-environment numbers. Headline behaviors, including the budget-degeneracy collapse, reproduced on freshly downloaded data with a fresh split, an unusually strong reproducibility check.

Full per-corpus tables for the equal-compute sweep and the `klauspost` baseline, the benchmark corpora, and the trainer itself are available at <REPO-URL-TODO> (commit <HASH-TODO>).

Table 1: Compression ratio (heldout, batch compression) by corpus and level across dictionary variants: no dictionary, `zstd -train default` (110KB), our trainer’s stage-1-only *tuned* variant, our trainer’s full pipeline, and a random raw-concatenation control. Δ is the ratio gain of *full* over *default*.

Corpus	Level	No-dict	Default	Tuned	Full (ours)	Raw-concat	Δ Full vs. Default
apijson	3	1.915	4.001	4.246	4.246	3.494	+6.11%
apijson	19	1.958	4.593	5.310	5.356	4.089	+16.62%
csvrows	3	4.342	6.190	6.613	6.606	5.392	+6.71%
csvrows	19	4.977	8.170	10.206	10.471	6.772	+28.17%
gharchive	3	3.741	7.680	8.443	8.403	6.665	+9.42%
gharchive	19	3.906	8.843	11.100	11.335	7.747	+28.18%
github_users	3	2.954	9.646	10.202	9.944	8.318	+3.09%
github_users	19	3.029	9.955	10.690	11.102	9.344	+11.52%
weblogs	3	3.521	6.421	7.320	7.343	5.270	+14.35%
weblogs	19	3.788	8.684	10.692	11.473	7.206	+32.12%

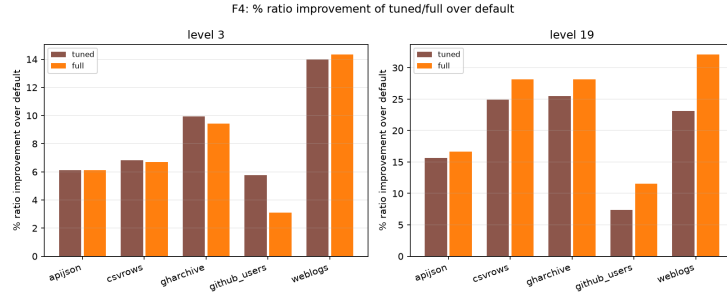


Figure 3: Percent compression-ratio improvement of *tuned* and *full* over the deployed default, by corpus and level (F4, generated by `tools/make_figures.py`).

7 Results

Table 1 and Figure 3 give the headline comparison against the deployed default; §7.1 states it plainly, §7.2 isolates how much of it is dictionary size versus method by rebuilding baselines at matched sizes and at production-motivated budgets, and §7.3 checks whether the gain survives when the incumbent is given equal wall-clock time and compares against the only other public dictionary trainer we found.

7.1 Main results

Against the deployed default, our trainer improves compression ratio on every corpus at both levels (Table 1): between +3.1% and +14.3% at level 3 and between +11.5% and +32.1% at level 19, averaging roughly +8% and +23% respectively (Figure 3). The no-training control – random training samples concatenated to the same 110 KiB – trails the stock trainer everywhere, confirming the covering heuristic earns its keep. The *tuned* column is Stage 1 alone; *full* adds Stages 2–3, contributing +1.0 to +9.0 percentage points at level 19, positive on all five corpora. At level 3, where Stages 2–3 are gated off, *tuned* and *full* should coincide exactly; where they instead differ, by 0% to -2.7% and most visibly on `github_users` (-2.53%), the cause is a validation-gate contamination in the full-refit step (§5.1) rather than genuine stage-2/3 contribution – a fix is in progress and every number marked *pending* in this section will change once it lands.

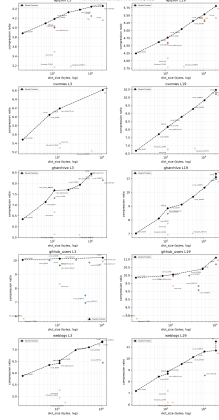


Figure 4: Compression ratio vs. dictionary size across every trainer/size variant we ran, one panel per (corpus, level); our *tuned* and *full* variants sit on or near the Pareto frontier in every panel (F3, generated by `tools/make_figures.py`).

7.2 Isolating size from method

Our trainer chooses its own size, usually larger than the 110 KiB default, so we rebuilt the baseline at *our* winning size for every corpus and level (Table 2, Figure 4). At level 3, almost none of the headline advantage survives – two corpora match exactly, two retain $+0.2\%/+1.5\%$, so 0–11% of the gain is method rather than size. At level 19 a real margin remains: $+6.2\%/+6.9\%/+2.8\%$ (`github_users/weblogs/csvrows`) at identical budgets. This defines the contribution: at fast levels the trainer *finds* the right size safely and automatically, a size the stock trainer could in principle also have produced; at high levels refinement and seeding add compression no choice of budget recovers. Three of ten matched-size cells could not be run as intended – the stock trainer, asked for our winning budget, itself emitted substantially less – so the size confound is not fully separable from §4.1.

Reproducing Cassandra’s shipped default and two further deployment-motivated settings (§3.2): at Cassandra’s 64KiB level-3 budget we beat the stock trainer on three corpora ($+0.4\%$ to $+1.7\%$) and tie exactly on two, no losses – ties occur where no ladder candidate beats the incumbent-equivalent floor, the anytime guarantee (§5.3) operating as designed. At a level-1, 110 KiB setting motivated by ScyllaDB’s (off-by-default) RPC path, we win on all five corpora, $+1.6\%$ to $+8.1\%$. RocksDB’s opt-in, non-default no-trainer fallback ranks between no dictionary and the stock trainer in nine of ten cells and trails our pipeline by a median of 18%.

7.3 Equal compute and alternative trainers

Our trainer spends minutes to tens of minutes (41 s–59 min across corpora and levels) where `zstd -train` spends 0.2–3.2 s. The corresponding manual control gives the incumbent our wall-clock budget to sweep sizes, selecting by measured validation compression: this closes the gap at level 3 (1.5% above to 9.5% below our result, a wash) while at level 19 we still win on four of five corpora, $+1.6\%$ to $+8.5\%$, losing narrowly on `apijson` (-1.1%) – these equal-compute figures predate the refit-on-full fix above and will be replaced once corrected values are computed. The sweep also reproduced the budget-degeneracy pathology – several large-budget candidates emitted dictionaries 25–30% worse than at half the budget, surviving only because our harness selects by measured compression, not requested size (§4.1, M3) – the clearest demonstration that the safety property in §5.3 matters.

`klauspost/compress`’s `bulddict` [18], the closest prior art to our level-awareness claim,

Table 2: Matched-size and production-configuration parity comparisons (e.g. Cassandra CEP-54’s 64KB default, ScyllaDB’s L1 RPC dictionary, RocksDB’s finalize-raw fallback), from the parity campaign (`tools/run_parity.py`).

Corpus	Level	Variant	Dict. size (B)	Ratio	Note
apijson	1	default110_L1	112,640	3.634	reused MAIN 'default' dict from runs/campaign_v2/apijson/3/
apijson	1	ours_L1	65,536	3.693	trainer.py --target-level 1 --max-budget 2097152 --time-budg
apijson	3	default64	65,536	3.924	zstd -train -r train --maxdict=65536 -T8; Cassandra real de
apijson	3	default_matched	952,604	3.705	zstd -train -r train --maxdict=1048576 -T8 (S=MAIN/full dic
apijson	3	ours64	65,536	3.924	trainer.py --target-level 3 --max-budget 65536 --time-budget
apijson	3	rocksdb_finalize	113,831	3.795	ZDICT_finalizeDictionary(content= 113664B raw concat of ran
apijson	19	default_matched	952,604	4.749	zstd -train -r train --maxdict=1048583 -T8 (S=MAIN/full dic
apijson	19	rocksdb_finalize	113,831	4.410	ZDICT_finalizeDictionary(content= 113664B raw concat of ran
csvrows	1	default110_L1	112,640	5.773	reused MAIN 'default' dict from runs/campaign_v2/csvrows/3/
csvrows	1	ours_L1	262,144	6.240	trainer.py --target-level 1 --max-budget 2097152 --time-budg
csvrows	3	default64	65,536	6.057	zstd -train -r train --maxdict=65536 -T8; Cassandra real de
csvrows	3	default_matched	1,048,576	6.606	zstd -train -r train --maxdict=1048576 -T8 (S=MAIN/full dic
csvrows	3	ours64	65,536	6.057	trainer.py --target-level 3 --max-budget 65536 --time-budget
csvrows	3	rocksdb_finalize	113,810	5.394	ZDICT_finalizeDictionary(content= 113664B raw concat of ran
csvrows	19	default_matched	2,097,143	10.189	zstd -train -r train --maxdict=2097143 -T8 (S=MAIN/full dic
csvrows	19	rocksdb_finalize	113,813	6.843	ZDICT_finalizeDictionary(content= 113664B raw concat of ran
gharchive	1	default110_L1	112,640	6.953	reused MAIN 'default' dict from runs/campaign_v2/gharchive/
gharchive	1	ours_L1	262,144	7.215	trainer.py --target-level 1 --max-budget 2097152 --time-budg
gharchive	3	default64	65,536	7.500	zstd -train -r train --maxdict=65536 -T8; Cassandra real de
gharchive	3	default_matched	1,048,576	8.388	zstd -train -r train --maxdict=1048576 -T8 (S=MAIN/full dic
gharchive	3	ours64	65,536	7.543	trainer.py --target-level 3 --max-budget 65536 --time-budget
gharchive	3	rocksdb_finalize	113,825	7.134	ZDICT_finalizeDictionary(content= 113664B raw concat of ran
gharchive	19	default_matched	892,712	8.571	zstd -train -r train --maxdict=2097152 -T8 (S=MAIN/full dic
gharchive	19	rocksdb_finalize	113,825	8.158	ZDICT_finalizeDictionary(content= 113664B raw concat of ran
github_users	1	default110_L1	112,640	8.989	reused MAIN 'default' dict from runs/campaign_v2/github_us
github_users	1	ours_L1	32,768	9.494	trainer.py --target-level 1 --max-budget 2097152 --time-budg
github_users	3	default64	65,536	9.835	zstd -train -r train --maxdict=65536 -T8; Cassandra real de
github_users	3	default_matched	1,048,576	9.944	zstd -train -r train --maxdict=1048576 -T8 (S=MAIN/full dic
github_users	3	ours64	16,384	10.004	trainer.py --target-level 3 --max-budget 65536 --time-budget
github_users	3	rocksdb_finalize	113,803	9.644	ZDICT_finalizeDictionary(content= 113664B raw concat of ran
github_users	19	default_matched	1,048,575	10.453	zstd -train -r train --maxdict=1048575 -T8 (S=MAIN/full dic
github_users	19	rocksdb_finalize	113,806	10.301	ZDICT_finalizeDictionary(content= 113664B raw concat of ran
weblogs	1	default110_L1	112,640	5.769	reused MAIN 'default' dict from runs/campaign_v2/weblogs/3/
weblogs	1	ours_L1	262,144	6.050	trainer.py --target-level 1 --max-budget 2097152 --time-budg
weblogs	3	default64	65,536	6.278	zstd -train -r train --maxdict=65536 -T8; Cassandra real de
weblogs	3	default_matched	1,048,576	7.232	zstd -train -r train --maxdict=1048576 -T8 (S=MAIN/full dic
weblogs	3	ours64	65,536	6.306	trainer.py --target-level 3 --max-budget 65536 --time-budget
weblogs	3	rocksdb_finalize	113,802	5.973	ZDICT_finalizeDictionary(content= 113664B raw concat of ran
weblogs	19	default_matched	2,097,159	10.738	zstd -train -r train --maxdict=2097159 -T8 (S=MAIN/full dic
weblogs	19	rocksdb_finalize	113,810	8.100	ZDICT_finalizeDictionary(content= 113664B raw concat of ran

loses to the stock zstd default on all twenty cells we ran, emitting ~ 17 KiB on `github_users` regardless of a 1 MiB request; we report this without enthusiasm – it should have been a harder baseline to beat.

All results above are single-machine (Apple M4) measurements from the rebuilt post-wipe environment; throughput, bootstrap confidence intervals, and cross-codec transfer to Brotli and LZ4 were designed but not re-executed here, so we report none rather than reuse figures we could not recompute.

8 Conclusion

Diagnosis-first beats algorithm-first: a ~ 1 -second exact oracle changes the design space. A 2016 covering heuristic, unmodified, plus a validation-driven selection layer around it yields double-digit archival gains, gains that survive matched-size and equal-compute scrutiny, and guaranteed-safe defaults at every budget – while the safe-automation story at fast levels is, on its own honest terms, still worth shipping. Two things remain open: byte-identical reproducibility across reruns, an assumption the safety guarantees lean on, was not independently re-checked after the post-wipe rebuild, and x86 throughput replication is still pending. We have drafted, but

not yet upstreamed, four fixes to the incumbent trainer itself – charging its objective by requested rather than emitted capacity, scaling its dead-epoch tolerance, fixing the suppressed diagnostics, and shrinking dictionaries from the front instead of the back. Sharing Stage 1’s search across levels, and a speed-constrained mode that trades ratio for a bounded training budget, are the natural next steps given the training-time cost measured in §7.3.

References

- [1] A. Afroozeh, L. X. Kuffó, and P. Boncz. ALP: Adaptive lossless floating-point compression. *Proceedings of the ACM on Management of Data (PACMOD)*, 1(4):230:1–230:26, 2023. Presented at ACM SIGMOD 2024.
- [2] J. Alakuijala, A. Farruggia, P. Ferragina, E. Kliuchnikov, R. Obryk, Z. Szabadka, and L. Vandevenne. Brotli: A general-purpose data compressor. *ACM Transactions on Information Systems (TOIS)*, 37(1):4:1–4:30, 2019.
- [3] F. B. Add `compressionLevel` parameter to `ZDict_trainFromBuffer`. GitHub issue #3213, facebook/zstd, July 2022.
- [4] P. Bille, I. L. Gørtz, S. J. Puglisi, and S. R. Tarnow. Hierarchical relative Lempel-Ziv compression. In *Proceedings of the 21st International Symposium on Experimental Algorithms (SEA)*, volume 265 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.
- [5] P. Boncz, T. Neumann, and V. Leis. FSST: Fast random access string compression. *Proceedings of the VLDB Endowment (PVLDB)*, 13(12):2649–2661, 2020.
- [6] K. Casel, H. Fernau, S. Gaspers, B. Gras, and M. L. Schmid. On the complexity of the smallest grammar problem over fixed alphabets. *Theory of Computing Systems*, 65(2):344–409, 2021. Journal version of the ICALP 2016 paper “On the Complexity of Grammar-Based Compression over Fixed Alphabets”.
- [7] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and abhi shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.
- [8] Cloudflare, Inc. Shared dictionaries. Cloudflare Developer Docs, 2026.
- [9] Y. Collet and M. S. Kucherawy. Zstandard compression and the `application/zstd` media type. RFC 8878, Informational, Feb. 2021. Obsoletes RFC 8478.
- [10] Y. Collet, N. Terrell, W. F. Handte, D. Rozenblit, V. Zhang, K. Zhang, Y. Goldschlag, J. Lee, E. Gorokhovskiy, Y. Komornik, D. Riegel, S. Angelov, and N. Rotem. OpenZL: A graph-based model for compression. arXiv preprint arXiv:2510.03203, Oct. 2025. Extended companion version: arXiv:2605.09928, “OpenZL: Using Graphs to Compress Smaller and Faster”.
- [11] Google Chrome team. Chrome Platform Status: Compression dictionary transport. [chromestatus.com](https://chromestatus.com/feature/5124977788977152), feature 5124977788977152, 2026.
- [12] F. Handte, Y. Collet, and N. Terrell. 5 ways Facebook improved compression at scale with Zstandard. Facebook (Meta) Engineering Blog, Dec. 2018.
- [13] C. Hoobin, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *Proceedings of the VLDB Endowment (PVLDB)*, 5(3):265–273, 2011.
- [14] N. Johari. [help wanted, questions] improving dictionary training process. GitHub issue #4127, facebook/zstd, Aug. 2024. Opened 2024-08-15, closed 2024-12-09.
- [15] K. Liao, M. Petri, A. Moffat, and A. Wirth. Effective construction of relative Lempel-Ziv dictionaries. In *Proceedings of the 25th International Conference on World Wide Web (WWW)*, pages 807–816, Montréal, Québec, Canada, 2016. International World Wide Web Conferences Steering Committee. COVER / fastCOVER dictionary-construction algorithm underlying `zstd -train`.
- [16] X. Lin. Dictionary training performance anomaly, any answers? GitHub issue #1572, facebook/zstd, Apr. 2019.
- [17] P. Meenan and Y. Weiss. Compression dictionary transport. RFC 9842, Standards Track, Sept. 2025.
- [18] K. Post. `klauspost/compress`: Optimized compression packages for Go. GitHub repository, 2024. `dict/cmd/builddict` zstd dictionary trainer, cited for its `-zlevel` level-targeting flag; benchmarked as baseline E2.
- [19] Y. Shi, X. Zou, X. Chen, S. Jin, D. Tao, C. Deng, Y. Chen, and W. Xia. Machete: An efficient lossy floating-point compressor designed for time series databases. In *Proceedings of the 2024 Data Compression Conference (DCC)*, pages 532–541. IEEE, 2024.
- [20] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *Journal of the ACM (JACM)*, 29(4):928–951, 1982.
- [21] G. Ye, T. Wang, X. Liu, R. Zhou, and Q. Zhou. LZHV: Accelerating LZ77 compression algorithm with hash verification. In *Proceedings of the 2025 Data Compression Conference (DCC)*, pages 133–142. IEEE, 2025.