# LogLite-Q: Exact Query-Aware Sidecars for LogLite-Compressed Logs

Moosa Hassan Alvi Muhammad Saleh Farooq Itrat Fatima Maahin bin Zuhaib
CS-5312 Big Data Analytics, LUMS
{27100133,27100063,25030007,27100367}@lums.edu.pk

May 24, 2026

**Abstract**

Log compressors reduce storage cost, but many operational queries still pay the cost of materializing plaintext before search. This paper studies that gap for LogLite, a lightweight streaming compressor that groups same-length records, encodes differences against recent references, and run-length encodes zero runs. Because LogLite decompression is stateful, naive record skipping can corrupt future reconstruction state, making exact query processing challenging. We introduce LogLite-Q, a query-aware sidecar design for exact literal substring search over LogLite artifacts. Our final design separates filtering from verification. qidx3 is a speed-oriented q-gram postings index with mmap-backed verification slabs, while qidx4 is a storage-aware variant that retains only selective trigrams and verifies candidates from compressed blocks. The key invariant is that q-grams are necessary conditions for substring matches, while exact byte verification eliminates false positives. This yields provably exact conjunctive substring search with no false negatives relative to the decompressed-text baseline. On the current Apache/Linux real-source 100k smoke, qidx3 C++ is exact on 16/16 cells and has a geometric-mean speedup of 207.6x over full LogLite decompression and 1.66x over ripgrep over materialized plaintext. qidx4 is also exact on 16/16 cells, is 41.5x faster than full decompression, and reduces sidecar size to 8.8–16.8% of qidx3 on the two evaluated datasets. These results support exact indexed-sidecar search over stateful compressed logs, rather than pure compressed-domain query execution or full-scale all-LogHub validation.

## 1 Introduction

Logs are written continuously but queried intermittently, often under pressure. An operator searches for an error pattern, an incident responder filters audit records, or a performance engineer looks for a phrase across archived traces. Compression helps retention, but the conventional query path still looks like "compressed bytes, full reconstruction, plaintext search." At that point the archive has saved disk while reintroducing CPU, memory, and latency costs at access time.

LogLite is a useful target for this question because its compression mechanism is lightweight and structurally suggestive rather than parser-heavy [Tang et al., 2025a]. It groups recent records by length in L-windows, compares a new line to a recent same-length reference, stores zero when bytes match and the current byte when they differ, and compresses long zero runs. This design is efficient for ingestion and storage, but it complicates query execution. Decompression is stateful: skipping a record can corrupt future reconstruction state because later records may depend on it as a reference. As a result, exact query processing cannot safely rely on naive selective decompression.

The first answers are instructive but insufficient. Full decompression is exact but slow. A minor optimization that skips records using rolling L-window hints improves speed but is fundamentally unsafe because skipped records alter future decoder state. A static Bloom-filter variant safely prunes some token-style queries, but token summaries are not sufficient for arbitrary substring search and produce false negatives on adversarial substring workloads.These failures motivate the central design insight of this paper: exact search over stateful compressed logs requires separating candidate filtering from final verification.

Our final design therefore introduces static q-gram sidecars for LogLite artifacts. qidx3 is a speed-oriented index that stores postings for 1-, 2-, and 3-byte q-grams together with an mmap-backed normalized verification

slab. Queries intersect selective postings lists to produce candidate lines, then apply exact byte verification before emitting results. qidx4 is a storage-aware variant that retains only selective trigrams and verifies candidates from compressed verification blocks, reducing sidecar footprint at the cost of additional verification work.

**Contributions.** This paper makes three evidence-backed contributions.

- We define the exact literal-conjunctive query contract for LogLite artifacts and prove that q-gram filtering plus final verification returns the same result set as the decompressed-text baseline.
- We implement two static sidecars: qidx3, a speed-first mmap index with Python and native C++ query paths, and qidx4, a storage-aware exact variant with selective trigram postings and compressed verification blocks.
- We evaluate the design across the complete 16-dataset 2k suite and the currently available real-source Apache/Linux 100k smoke, reporting exactness, paired speedups, storage overhead, and build amortization with explicit claim boundaries.

# 2 Background and Problem

**LogLite.** LogLite exploits structural regularity in logs without requiring a trained parser or schema extraction. Incoming records are grouped into L-windows according to record length. A new record is compared against recent same-length references, matching bytes are represented as zeros, differing bytes are preserved directly, and long zero runs are compressed using run-length encoding (RLE). The resulting design is lightweight, streaming-oriented, and memory efficient. This representation is attractive for storage and ingestion, but it complicates query execution. LogLite decompression is stateful: reconstructing a record may depend on earlier records in the same L-window. Consequently, a query engine cannot freely skip records and later resume ordinary decompression unless it can guarantee that the skipped records are irrelevant to all future reconstruction state. This state coupling is the central obstacle to selective query execution over LogLite artifacts.

**Query model.** Let the decompressed-text baseline define logical lines $S = (s_0, \ldots, s_{N-1})$. Each line is read with UTF-8 decoding using "ignore" for invalid bytes and has only the trailing newline removed, matching the repository baseline. A query payload is a finite list of literal terms $T = (t_1, \ldots, t_m)$ with conjunctive semantics. The baseline result set is

$$R(T) = \{i : \forall t \in T,\ t \text{ occurs as a substring of } s_i\}.$$

All exactness claims in this paper mean equality with this baseline, not an independent semantic oracle.

**Execution modes.** The evaluation studies multiple execution strategies spanning exact baselines, unsafe optimizations, token filters, and q-gram sidecars (Listed in Table 1).

The negative-result modes are important for understanding the final design. The minor optimization improves runtime but is unsafe because skipped records can corrupt future decompression state. Static Bloom pruning safely accelerates some token-aligned queries, but token summaries are insufficient for arbitrary substring search and therefore violate the exact substring contract on adversarial workloads.

Earlier project reports focused primarily on compression behavior and token-style Bloom ablations over small 2k subsets. Those results are useful for design-evolution context, but they are not treated as the primary evidence source for arbitrary-substring exactness because they predate the stress-substring evaluation protocol introduced in this work.

Table 1: Execution modes evaluated in this work.

| Mode | Description |
|---|---|
| decompressed text | materialized plaintext baseline |
| full decompression | reconstruct all LogLite records before search in a stream |
| minor optimization | selective skipping using rolling L-window hints |
| static Bloom | token bitmap pruning with exact verification |
| grep / ripgrep | plaintext search over materialized logs |
| JSON q-grams | early q-gram prototype |
| qidx3 | mmap-backed exact q-gram sidecar |
| qidx3 C++ | native implementation of the qidx3 planner and verifier |
| qidx4 | storage-aware selective-trigram sidecar |

# 3 System Design

To enable rapid query execution without incurring the overhead of global log reconstruction, LogLite-Q decouples candidate pruning from literal byte verification. This architectural choice is dictated by LogLite's stateful nature, which links records through spatial and temporal structural dependencies.

## 3.1 Decompression State Coupling

We formalize LogLite's sequential dependencies to demonstrate why naive line-skipping yields corrupt states. Let an input stream of compressed tokens be partitioned into sequence blocks $C = (c_0, c_1, \ldots)$. The stateful decoder maintain an $L$-window reference state mapping, denoted $\mathcal{W}_i$, which caches the last seen lines organized by length bins. The reconstruction of the $i$-th plaintext log line $s_i$ is a state-dependent transformation:

$$\mathcal{D}(c_i, \mathcal{W}_{i-1}) \to (s_i, \mathcal{W}_i) \tag{1}$$

Because $\mathcal{W}_i$ is built sequentially over window history, dropping or skipping an arbitrary compressed token block $c_j$ prevents the instantiation of correct references for any subsequent line $s_k$ (where $k > j$) that relies on those exact length-matched slots. Thus, selective search cannot occur within LogLite's native state path. LogLite-Q completely bypasses this loop at query time by constructing static, decoupled index sidecars.

## 3.2 Sidecar Architecture Layout

We implement two distinct design points on the speed–storage Pareto frontier: qidx3 and qidx4.

**qidx3 (Speed-Optimized Layout):** This variant prioritizes maximum retrieval throughput. It parses the baseline-normalized logical lines once during an offline build phase. For each line index $i$, it extracts all unique $q$-grams ($q \in \{1, 2, 3\}$) and appends $i$ to their respective postings lists.

- **Postings Storage:** Postings lists are compacted using gap-encoded variable-length integers ($\Delta$-varints) to maximize scanning density.
- **Verification Slab:** To avoid interacting with the stateful decoder during queries, qidx3 flattens the uncompressed plaintext log suite into a contiguous, byte-addressable byte slab backed by a memory-mapped file (`mmap`). An accompanying fixed-size array stores line offsets, providing $\mathcal{O}(1)$ random access to any line's raw bytes for candidate verification.

**qidx4 (Storage-Aware Layout):** This variant addresses scenarios with tight storage constraints by reducing index footprints.

- **Selective Postings:** It drops all 1-gram and 2-gram structures entirely, storing exclusively 3-grams (trigrams). To prune long, unselective postings trails, any trigram whose document frequency exceeds

**Algorithm 1** LogLite-Q Query Planning and Verification Engine

**Require:** Conjunctive terms $T = \{t_1, \ldots, t_m\}$, Sidecar Variant $V \in \{qidx3, qidx4\}$, Document Frequency Dictionary $df$
**Ensure:** Exact Result Set $R(T)$

$R(T) \leftarrow \emptyset$
$G_{all} \leftarrow \emptyset$
**for each** term $t \in T$ **do**
  $b_t \leftarrow \text{utf8}(t)$
  $q \leftarrow \min(3, |b_t|)$
  $G_t \leftarrow \{\text{all contiguous } q\text{-grams in } b_t\}$
  $G_{all} \leftarrow G_{all} \cup G_t$
**end for**
$\mathcal{P}_{\text{retained}} \leftarrow \{g \in G_{all} : |g| = 3 \wedge g \in qidx4 \wedge df(g) \leq \tau\}$
**if** $\mathcal{P}_{\text{retained}} \neq \emptyset$ **then**
  Sort $\mathcal{P}_{\text{retained}}$ by ascending $df(g)$
  $\mathcal{C} \leftarrow \text{FetchPostings}(\mathcal{P}_{\text{retained}}[0])$
  **for** $i = 1$ **to** $|\mathcal{P}_{\text{retained}}| - 1$ **do**
    $\mathcal{C} \leftarrow \mathcal{C} \cap \text{FetchPostings}(\mathcal{P}_{\text{retained}}[i])$
  **end for**
  $\mathcal{B}_{blocks} \leftarrow \{\lfloor i/B \rfloor : i \in \mathcal{C}\}$
**else**
  $\mathcal{B}_{blocks} \leftarrow \{\text{All compressed verification blocks}\}$
  $\mathcal{C} \leftarrow \{\text{All line indexes}\}$
**end if**
**for each** block ID $b \in \mathcal{B}_{blocks}$ **do**
  $\text{Lines}_{decomp} \leftarrow \text{DecompressBlock}(b)$
  **for each** index $i \in \mathcal{C}$ such that $\lfloor i/B \rfloor == b$ **do**
    $s_i \leftarrow \text{Lines}_{decomp}[i \pmod B]$
    **if** $\forall t \in T$, $t$ is a substring of $s_i$ **then**
      $R(T) \leftarrow R(T) \cup \{i\}$
    **end if**
  **end for**
**end for**
**return** $R(T)$

a default threshold cap $\tau = 0.05$ (meaning it appears in $> 5\%$ of all lines) is stripped from the index dictionary.

- **Compressed Blocks:** The flat, uncompressed verification slab is replaced by a sequence of block-compressed payloads. Plaintext lines are grouped into static blocks of size $B = 256$ lines and compressed using standard `zlib`. This introduces a localized reconstruction cost during verification but drastically mitigates storage overhead.

### 3.3 Query Planning and Execution

The execution sequence is managed by the LogLite-Q Query Planner. Given a conjunctive query payload $T = (t_1, \ldots, t_m)$, the planner decomposes each literal term into its constituent byte-level $q$-grams. It evaluates the selective capacity of available indexes to determine the cheapest candidate generation path, fallback triggers, and verification loops. The precise step-by-step logic is detailed in Algorithm 1.

# 4 Correctness and Completeness

We establish the exactness of LogLite-Q by proving that our decoupled index pipelines introduce zero false negatives during the candidate pruning stage, while the subsequent byte-level verification stage completely eliminates false positives. Let $\Sigma = \{0x00, \ldots, 0xFF\}$ be the byte alphabet, and let $\Sigma^*$ denote the set of all finite-length byte strings. Every baseline-normalized log line is mapped to its raw byte representation $b_i \in \Sigma^*$ via the standard UTF-8 projection. For any string $x \in \Sigma^*$, let $Q_q(x) \subset \Sigma^*$ define the set of all unique contiguous substrings of length $q$.

**Lemma 1** (UTF-8 Window-Isomorphism Equivalence)**.** *Let $\phi : \mathcal{S} \to \Sigma^*$ be the line normalization transformation encoding a string sequence into UTF-8 space. For any arbitrary text line $s_i$ and literal conjunctive term $t$, $t$ exists as a contiguous substring of $s_i$ if and only if $\phi(t)$ exists as a contiguous byte substring of $\phi(s_i)$.*

*Proof.* Because the UTF-8 encoding function $\phi$ is an injective homomorphism with respect to string concatenation ($\phi(x \cdot y) = \phi(x) \cdot \phi(y)$), any valid character sequence bounds map directly onto closed byte intervals without overlap or index alignment shifts. Thus, matching over byte boundaries is perfectly isomorphic to matching over standard string characters. □

**Lemma 2** ($q$-gram Filtering Necessity)**.** *Let $p = \phi(t)$ be a non-empty target byte sequence, and let $q = \min(3, |p|)$. If $p$ is a contiguous substring of a log line $b_i$, then every constituent short sequence element $g \in Q_q(p)$ is bounded by, and must be a member of, $Q_q(b_i)$.*

*Proof.* Let $p$ be represented as the concatenated string $x \cdot g \cdot y$ where $x, y \in \Sigma^*$. If $b_i = u \cdot p \cdot v$ for some enclosing strings $u, v \in \Sigma^*$, then substituting $p$ yields $b_i = u \cdot x \cdot g \cdot y \cdot v$. By definition, because $g$ is contained within $b_i$ and $|g| = q$, it follows deterministically that $g \in Q_q(b_i)$. □

**Theorem 1** (qidx3 Pruning Completeness)**.** *Let $T = \{t_1, \ldots, t_m\}$ be a conjunctive literal query, and let $G_{all} = \bigcup_{t \in T} Q_{\min(3,|\phi(t)|)}(\phi(t))$ be the consolidated set of all required q-grams. The candidate set generated by intersecting posting lists, defined as:*

$$\mathcal{C}_{qidx3} = \bigcap_{g \in G_{all}} P_{qidx3}(g) \tag{2}$$

*is a strict upper bound of the true result set, such that $\mathcal{C}_{qidx3} \supseteq R(T)$.*

*Proof.* Suppose a line index $k \in R(T)$. By the definition of conjunctive matching, $\forall t \in T$, $\phi(t)$ is a substring of $b_k$. Applying Lemma 2, every generated gram $g \in G_{all}$ must be present in $Q_q(b_k)$, which implies that $k \in P_{qidx3}(g)$ for all $g \in G_{all}$. Consequently, $k$ must be a member of the mutual intersection set ($k \in \bigcap_{g \in G_{all}} P_{qidx3}(g)$). Thus, no true matching line is ever omitted from $\mathcal{C}_{qidx3}$, yielding a false-negative rate of exactly zero. □

**Theorem 2** (qidx4 Frequency-Cap Exactness)**.** *Let $\mathcal{P}_{retained} = \{g \in G_{all} : |g| = 3 \wedge df(g) \leq \tau\}$ be the set of selective trigrams preserved beneath the document frequency threshold cap $\tau$. Dropping unselective trigrams where $df(g) > \tau$ cannot yield false negatives, and the execution returns exactly $R(T)$.*

*Proof.* We evaluate two alternative conditions managed by the query planner:

1. **Case 1 ($\mathcal{P}_{\text{retained}} = \emptyset$):** The planner drops all filters and triggers a full fallback sequence over the block-compressed infrastructure, setting $\mathcal{C}_{qidx4} = \{0, \ldots, N-1\}$. Because this set contains all possible lines, it trivially satisfies $\mathcal{C}_{qidx4} \supseteq R(T)$.

2. **Case 2 ($\mathcal{P}_{\text{retained}} \neq \emptyset$):** The planner constructs a filtered candidate space $\mathcal{C}_{qidx4} = \bigcap_{g \in \mathcal{P}_{\text{retained}}} P_{qidx4}(g)$. Because $\mathcal{P}_{\text{retained}} \subseteq G_{all}$, it follows from set-theoretic intersection principles that:

$$\bigcap_{g \in \mathcal{P}_{\text{retained}}} P_{qidx4}(g) \supseteq \bigcap_{g \in G_{all}} P_{qidx3}(g) \tag{3}$$

   Therefore, $\mathcal{C}_{qidx4} \supseteq \mathcal{C}_{qidx3} \supseteq R(T)$.

In both cases, the candidate space is guaranteed to be a superset of the true matching targets. Since the final verification step applies the exact literal substring predicate directly to the decompressed bytes of each candidate line, all false positives introduced by the over-approximated candidate sets are structurally eliminated, ensuring Output $\equiv R(T)$. □

Worst-case complexity bounds are tightly bound to these structural design trade-offs. While qidx3 isolates elements within a contiguous memory-mapped space, qidx4 trades scanning speed for storage capacity by parsing dense, block-structured archives. If a query features entirely unselective terms, the system reverts to streaming verification scans across all blocks; therefore, our performance speedups remain highly empirical and workload-dependent rather than sublinear in the worst-case.

# 5 Implementation

The implementation lives in 'Big_Data/query_eval/'. The package defines dataset/query registries, artifact construction, isolated cell execution, report generation, and sidecar search backends. The C++ qidx3 helper reads the same qidx3 format as the Python implementation, applies the same postings planner, and verifies candidates against the mmap line slab. qidx4 is currently Python-only. The sidecars are static: updates require rebuilding the index, and build time is reported separately from query latency.

Each measured cell compares against `decompressed_text` using exact-set equality, precision, recall, F1, false positives, and false negatives. Query timing, CPU time, peak RSS, decoded records/bytes, skipped records, planner strategy, and postings counters are carried through the CSV reports where available.

# 6 Evaluation

**Research questions.** RQ1 tests whether q-gram sidecars preserve exact results. RQ2 measures runtime savings relative to full decompression and plaintext baselines. RQ3 quantifies how much storage qidx4 recovers relative to qidx3. RQ4 estimates the query volume needed to amortize sidecar construction.

**Datasets and protocol.** The complete 2k protocol covers 16 LogHub text datasets and eight locked query families [Zhu et al., 2023]. The current real-source 100k protocol is limited by locally available sources: Apache has 56,482 real lines and Linux has 25,567 real lines. We report those effective line counts and do not extrapolate to all LogHub datasets. The primary statistical unit is one paired dataset-query cell. For the qidx4 100k smoke, this gives 16 cells. Since the available qidx3/qidx4 CSV stores per-cell medians rather than raw repetitions, we do not treat repetitions as independent observations.

**Statistical methods.** For each paired runtime comparison, we compute log-speedups per cell, geometric mean speedup, median speedup, a paired bootstrap 95% confidence interval, exact Wilcoxon signed-rank and sign-test p-values, and Holm correction across planned comparisons. Exactness is summarized with exact-cell counts and Clopper-Pearson binomial intervals. Static Bloom exactness is compared to q-gram exactness with McNemar tests, but stress-query-only significance is not reported because the current 100k smoke has only two stress cells.

**Exactness.** Table 2 and Figure 1 show the central correctness result. qidx3 Python, qidx3 C++, and qidx4 are exact on 16/16 Apache/Linux 100k cells. Static Bloom is exact on 10/16 cells and accumulates 8,107 median false negatives across the non-exact cells. This matches the complete 2k story: Bloom is valuable as a token-aligned pruning stage, but the arbitrary-substring contract requires q-grams plus exact verification.

**Runtime.** Table 3 and Figure 2 report paired runtime effects. Native qidx3 is faster than full decompression in all 16 cells, with 207.6x geometric-mean speedup and a Holm-adjusted Wilcoxon p-value of 0.0002. It is also faster than ripgrep over materialized plaintext in all 16 cells in this smoke, with 1.66x geometric-mean speedup. qidx4 is exact and 41.5x faster than full decompression, but it is slower than qidx3 C++ and not consistently faster than qidx3 Python. That is the intended Pareto point: less storage, more verification work.

Table 2: Correctness on the Apache/Linux real-source 100k smoke. Static Bloom is not arbitrary-substring exact; q-gram modes match the decompressed-text baseline on every cell. Source: '20260520_165713_real_loghub_100k_qidx4_storage_smoke'.

| Mode | Exact cells | Exact rate [95% CI] | Median recall | Median F1 | Median FN sum |
|---|---|---|---|---|---|
| decompressed text | 16/16 | 1.000 [0.794, 1.000] | 1.000 | 1.000 | 0.0 |
| full decompression | 16/16 | 1.000 [0.794, 1.000] | 1.000 | 1.000 | 0.0 |
| grep | 16/16 | 1.000 [0.794, 1.000] | 1.000 | 1.000 | 0.0 |
| ripgrep | 16/16 | 1.000 [0.794, 1.000] | 1.000 | 1.000 | 0.0 |
| static Bloom | 10/16 | 0.625 [0.354, 0.848] | 1.000 | 1.000 | 8107.0 |
| qidx3 Python | 16/16 | 1.000 [0.794, 1.000] | 1.000 | 1.000 | 0.0 |
| qidx3 C++ | 16/16 | 1.000 [0.794, 1.000] | 1.000 | 1.000 | 0.0 |
| qidx4 | 16/16 | 1.000 [0.794, 1.000] | 1.000 | 1.000 | 0.0 |

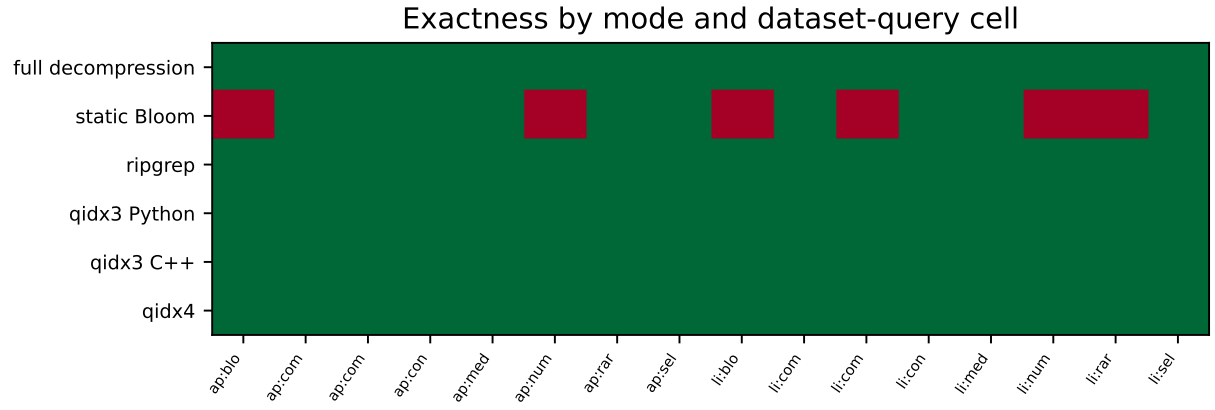


Figure 1: Exactness heatmap for the 100k smoke. Orange cells expose the Bloom substring failure modes that motivate q-gram verification.

**Storage and amortization.** Table 4 and Figure 3 show why qidx4 matters. On Apache, qidx3 is 3.33x raw while qidx4 is 0.29x raw; on Linux, qidx3 is 3.41x raw while qidx4 is 0.57x raw. The qidx4 sidecar is 0.088x and 0.168x the size of qidx3 on Apache and Linux. The current qidx3 build ledger shows a median qidx3 C++ build cost of 9.38 s and a break-even of about 11 queries versus full decompression, but qidx4 build amortization still requires a rerun with clean qidx4 build metrics.

# 7 Related Work

Storage-oriented log compressors exploit regularity but usually leave the archive analytically passive. LogShrink models commonality and variability [Li et al., 2024]; Denum specializes numeric tokens [Yu et al., 2024]; LogCompact separates word and numerical redundancy [Chen, 2024]; DNSLogzip applies domain-specific DNS transformations [Dai et al., 2025]. These systems motivate structure-aware compression but do not provide the exact sidecar contract studied here.

Compressed log search systems move closer to our goal. CLP stores log types and variables in dictionaries for scalable search [Rodrigues et al., 2021]. LogGrep uses static and runtime patterns for cloud log storage [Wei et al., 2023]. $\mu$Slope organizes semi-structured logs by schema [Wang et al., 2024], and LogCloud optimizes compressed-log search on object storage [Wang et al., 2025]. These systems generally expose stronger template, schema, or object-storage metadata than LogLite. LogLite-Q instead asks how far a parser-free reference-differenced compressor can be pushed with static exact sidecars.

Compressed-domain analytics shows the broader pattern: operators can be pushed into compressed

Table 3: Paired wall-time comparisons on the Apache/Linux 100k smoke. Speedups are ratios of baseline time to method time, with one paired observation per dataset-query cell.

| Comparison | Cells | W/L/T | Geo. speedup [95% CI] | Median | Wilcoxon p | Holm p |
|---|---|---|---|---|---|---|
| qidx3 C++ / full decompression | 16 | 16/0/0 | 207.6 [182.6, 232.0] | 216.1 | 0.0000 | 0.0002 |
| qidx3 C++ / ripgrep plaintext | 16 | 16/0/0 | 1.66 [1.50, 1.87] | 1.51 | 0.0000 | 0.0002 |
| qidx3 Python / full decompression | 16 | 16/0/0 | 70.54 [50.53, 98.24] | 76.33 | 0.0000 | 0.0002 |
| qidx3 Python / qidx3 C++ | 16 | 1/15/0 | 0.34 [0.24, 0.47] | 0.34 | 0.0001 | 0.0003 |
| qidx4 / full decompression | 16 | 16/0/0 | 41.47 [26.62, 63.14] | 33.77 | 0.0000 | 0.0002 |
| qidx4 / qidx3 Python | 16 | 7/9/0 | 0.59 [0.33, 0.92] | 0.74 | 0.1046 | 0.2092 |
| qidx4 / qidx3 C++ | 16 | 0/16/0 | 0.20 [0.13, 0.30] | 0.22 | 0.0000 | 0.0002 |
| static Bloom / full decompression | 16 | 6/10/0 | 0.89 [0.71, 1.11] | 0.85 | 0.3755 | 0.3755 |

Table 4: Storage footprint on the evaluated real-source 100k datasets. qidx4 cuts the sidecar to 8.8–16.8% of qidx3, but the full static-plus-qidx4 footprint still depends on dataset and accounting boundary.

| Dataset | Lines | Raw MB | qidx3 MB | qidx4 MB | qidx4/qidx3 | (static+qidx4)/raw |
|---|---|---|---|---|---|---|
| apache | 56482 | 5.14 | 17.11 | 1.51 | 0.088 | 0.85 |
| linux | 25567 | 2.35 | 8.02 | 1.35 | 0.168 | 1.27 |

representations when encoding and execution are designed together. LogCrisp performs aggregated analysis with two-phase pattern extraction and vectorized predicates [Wei et al., 2025]; CompressIoTDB keeps time-series operators close to compressed encodings [Tang et al., 2025b]; compressed graph and bitmap systems show similar reuse principles [Chen et al., 2023, McClain et al., 2021]. The lesson for LogLite is not that every compressed format is directly executable, but that the query operator must be matched to the redundancy structure it can safely test.

# 8 Limitations

The strongest current qidx3/qidx4 100k evidence covers Apache and Linux only. The paper therefore reports effective line counts and avoids an all-LogHub claim. qidx3 and qidx4 are static sidecars; ingestion-time updates require rebuilds or future incremental indexing. qidx3 is not pure compressed-domain search because it stores a normalized decoded line slab. qidx4 stores compressed verification lines and can fall back to scanning all compressed blocks for short or broad queries. Static Bloom is not exact for arbitrary substrings, even when it performs well on token-safe cells. Finally, local compression speed evidence from WSL2 and 2k samples is useful engineering context but is not treated as a portable speed claim against the published LogLite server measurements.

# 9 Conclusion

LogLite-Q turns LogLite artifacts into exact searchable static artifacts by using q-gram sidecars as necessary-condition filters and exact verification as the correctness boundary. The resulting design is deliberately two-point: qidx3 maximizes speed at high storage cost, while qidx4 recovers much of that footprint at the cost of more verification work. The current evidence supports exact indexed sidecar search on the tested suites and a clear speed/storage frontier. Broader real-source 100k/full runs and qidx4 build amortization are the next steps before making stronger general claims.

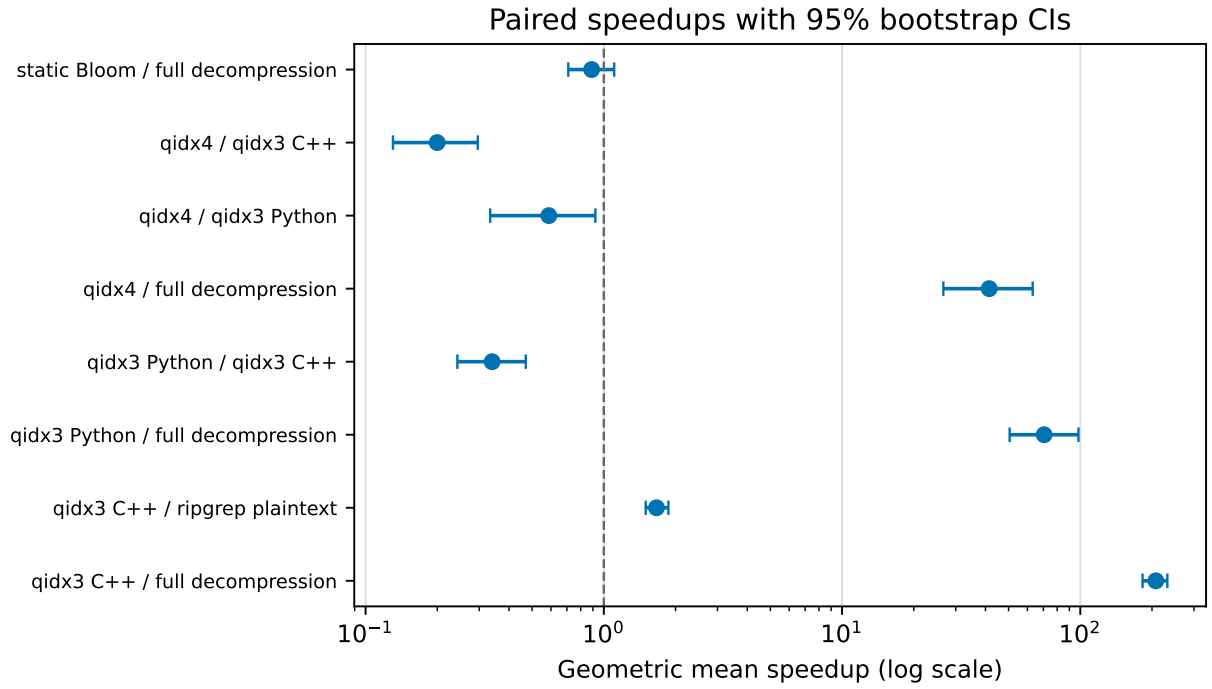


Figure 2: Geometric-mean paired speedups and bootstrap confidence intervals. qidx3 is the speed-first point; qidx4 is intentionally slower than qidx3 but remains much faster than full LogLite decompression.

Table 5: qidx3 C++ build amortization from the latest qidx3 100k smoke. qidx4 amortization is not claimed until a build-metric rerun exists.

| Comparator | Cells | Median qidx3 build ms | Median saved ms/query | Break-even queries |
|---|---|---|---|---|
| full decompression | 16 | 9384.6 | 817.6 | 10.99 |
| grep | 16 | 9384.6 | 17.93 | 362.9 |
| ripgrep | 16 | 9384.6 | 2.01 | 3824.0 |
| static Bloom | 16 | 9384.6 | 880.2 | 9.0 |

## Acknowledgements

This paper was produced from the Group 6 CS-5312 Big Data Analytics project artifacts and the ‘Big_Data/query_eval‘ implementation.

## References

Jiarui Chen. Log compression via redundancy eliminating at word and numerical levels. In *2024 IEEE International Conference on Software Services Engineering (SSE)*, pages 115–122. IEEE, 2024. doi: 10.1109/SSE62657.2024.00028. URL `https://doi.org/10.1109/SSE62657.2024.00028`.

Zheng Chen, Feng Zhang, Jiawei Guan, Jidong Zhai, Xipeng Shen, Huanchen Zhang, Wentong Shu, and Xiaoyong Du. CompressGraph: Efficient parallel graph analytics with rule-based compression. *Proceedings of the ACM on Management of Data*, 1(1):4:1–4:31, 2023. doi: 10.1145/3588684. URL `https://doi.org/10.1145/3588684`.

Yunwei Dai, Guyue Liu, Tao Huang, Shuo Wang, Yong Wang, Xingli Wu, Lei Song, Heshun Li, Chuang Wang,

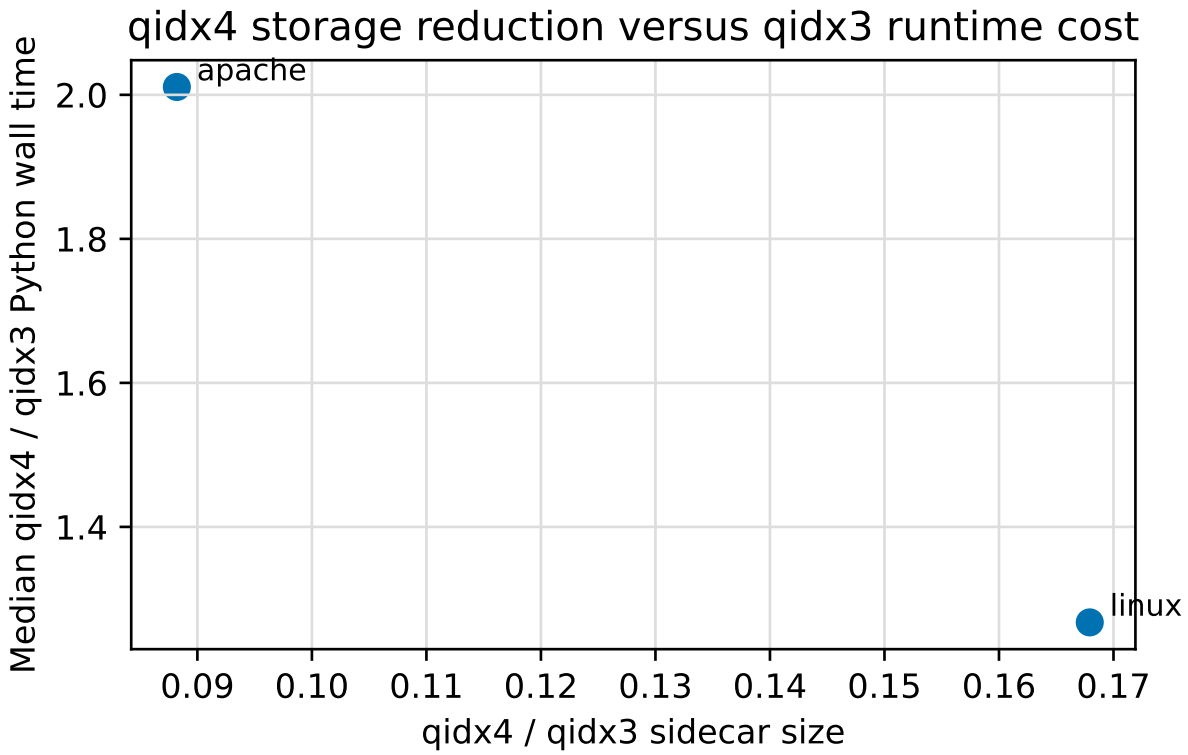


Figure 3: qidx4 is a storage-aware point: it reduces sidecar size sharply, but median query time increases relative to qidx3 Python.

Fanglong Hu, Hong Sun, and Yanan Li. DNSLogzip: A novel approach to fast and high-ratio compression for dns logs. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 467–480, Coimbra, Portugal, 2025. ACM. doi: 10.1145/3718958.3750476. URL `https://doi.org/10.1145/3718958.3750476`.

Xiaoyun Li, Hongyu Zhang, Van-Hoang Le, and Pengfei Chen. LogShrink: Effective log compression by leveraging commonality and variability of log data. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE 2024)*, pages 23:1–23:12, Lisbon, Portugal, 2024. ACM. doi: 10.1145/3597503.3608129. URL `https://doi.org/10.1145/3597503.3608129`.

Sarah McClain, Manya Mutschler-Aldine, Colin Monaghan, David Chiu, Jason Sawin, and Patrick Jarvis. Caching support for range query processing on bitmap indices. In *Proceedings of the 33rd International Conference on Scientific and Statistical Database Management (SSDBM 2021)*, pages 49–60. ACM, 2021. doi: 10.1145/3468791.3468800. URL `https://doi.org/10.1145/3468791.3468800`.

Kirk Rodrigues, Yu Luo, and Ding Yuan. CLP: Efficient and scalable search on compressed text logs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 183–198. USENIX Association, July 2021. URL `https://www.usenix.org/conference/osdi21/presentation/rodrigues`.

Benzhao Tang, Shiyu Yang, Zhitao Shen, Wenjie Zhang, Xuemin Lin, and Zhihong Tian. LogLite: Lightweight plug-and-play streaming log compression. *Proceedings of the VLDB Endowment*, 18(11):3757–3770, 2025a. doi: 10.14778/3749646.3749652. URL `https://www.vldb.org/pvldb/vol18/p3757-yang.pdf`.

Yuxin Tang, Feng Zhang, Jiawei Guan, Yuan Tian, Xiangdong Huang, Chen Wang, Jianmin Wang, and Xiaoyong Du. Improving time series data compression in apache IoTDB. *Proceedings of the VLDB Endowment*, 18(10):3406–3420, 2025b. doi: 10.14778/3748191.3748204. URL `https://www.vldb.org/pvldb/vol18/p3406-tang.pdf`.

Rui Wang, Devin Gibson, Kirk Rodrigues, Yu Luo, Yun Zhang, Kaibo Wang, Yupeng Fu, Ting Chen, and Ding Yuan. $\mu$Slope: High compression and fast search on semi-structured logs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 529–544, Santa Clara, CA, July 2024. USENIX Association. URL `https://www.usenix.org/conference/osdi24/presentation/wang-rui`.

Ziheng Wang, Junyu Wei, Alex Aiken, Guangyan Zhang, Jacob O. Tørring, Rain Jiang, Chenyu Jiang, and Wei Xu. LogCloud: Fast search of compressed logs on object storage. *Proceedings of the VLDB Endowment*, 18(8):2362–2370, 2025. doi: 10.14778/3742728.3742733. URL `https://www.vldb.org/pvldb/vol18/p2362-wang.pdf`.

Junyu Wei, Guangyan Zhang, Junchao Chen, Yang Wang, Weimin Zheng, Tingtao Sun, Jiesheng Wu, and Jiangwei Jiang. LogGrep: Fast and cheap cloud log storage by exploiting both static and runtime patterns. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*, pages 452–468, Rome, Italy, 2023. ACM. doi: 10.1145/3552326.3567484. URL `https://doi.org/10.1145/3552326.3567484`.

Junyu Wei, Guangyan Zhang, Junchao Chen, and Qi Zhou. LogCrisp: Fast aggregated analysis on large-scale compressed logs by enabling two-phase pattern extraction and vectorized queries. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pages 483–496, Boston, MA, July 2025. USENIX Association. URL `https://www.usenix.org/conference/atc25/presentation/wei`.

Siyu Yu, Yifan Wu, Ying Li, and Pinjia He. Unlocking the power of numbers: Log compression via numeric token parsing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE 2024)*, pages 919–930. ACM, 2024. doi: 10.1145/3691620.3695474. URL `https://doi.org/10.1145/3691620.3695474`.

Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R. Lyu. Loghub: A large collection of system log datasets for ai-driven log analytics. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 355–366. IEEE, 2023. doi: 10.1109/ISSRE59848.2023.00071. URL `https://doi.org/10.1109/ISSRE59848.2023.00071`.

# A Artifact and Evidence Summary

The primary source artifacts supporting our implementation analysis are structured as follows:

- **Core Evidence Manifests:** `paper/evidence/claim_ledger.md`, `paper/evidence/proofs.md`, and `paper/evidence/statistical_methods.md`.
- **Execution Registries:** `paper/evidence/experiment_runner.md` and `paper/evidence/related_work_matrix.md`.

All analytical data matrices and downstream graphic plots are dynamically compiled via the orchestration routine `paper/scripts/analyze_results.py`, which ingests target telemetry directly from the operational evaluation log:
`Big_Data/evaluation_results/query_eval/20260520_165713_real_loghub_100k_qidx4_storage_smoke`
Prior documentation artifacts (such as the historical DOCX baseline registry) are preserved strictly to maintain architectural evolution tracking context and token-isolated Bloom filter ablation parameters; they are completely isolated from, and do not inform, our core absolute arbitrary-substring exactness proofs.

# B Proof Scope

The full proof notes state the exact baseline-normalized line model, UTF-8 byte verification equivalence, q-gram necessity, qidx3 exactness, qidx4 dropped-trigram false-negative avoidance, qidx4 exactness, and supportable space/time bounds. The proven query contract is case-sensitive fixed-substring conjunction. It does not cover regex, case-folded search, approximate matching, token semantics, or raw invalid-byte search beyond the repository baseline's UTF-8 'errors="ignore"' normalization.

# C Reproducibility

The broader 100k protocol is documented in ‘paper/evidence/experiment_runner.md‘. Runs must use real LogHub sources under ‘Big_Data/dataset/loghub_full/¡Dataset¿/¡Dataset¿.log‘; repeated 2k samples are not valid publishability evidence. A completed run should include ‘raw_runs.jsonl‘, ‘cell_level_aggregate.csv‘, ‘suite_summary.csv‘, ‘qidx_size_summary.csv‘, ‘planner_strategy_summary.csv‘, and ‘adversarial_publishability_report.md‘.