CtrlB: Elastic Full-Text Search Over Petabyte-Scale Logs on Object Storage

Adarsh Srivastava
CtrlB Explore Inc.
Bengaluru, India
adarsh@ctrlb.ai

Abstract

Log search is a uniquely challenging workload: it demands both the analytical capabilities of columnar engines—aggregation, histograms, percentiles over structured fields—and the full-text search capabilities of inverted-index engines—substring matching, keyword lookup, ad-hoc pattern discovery over unstructured message bodies. These are competing philosophies: analytical queries benefit from columnar storage that keeps each field contiguous for vectorized scans, while full-text search benefits from row-oriented inverted indexes that map terms to individual records. Existing systems commit to one side, forcing operators to either sacrifice search quality (columnar engines) or pay enormous storage costs (inverted-index engines).

We present CtrlB, a log search and analytics system that resolves this tension through a novel indexing architecture built on columnar storage. CtrlB stores raw data as Parquet files on object storage (S3/GCS), with ingestion, indexing, querying, and compaction running as fully decoupled, independently scalable compute units. CtrlB introduces two index structures: the *Rare Term Index* (RTI), a per-file sidecar index that adapts its storage strategy based on term frequency, and the *Secondary File-level Skip Index* (SFS), a bloom-filter-based structure for file-level pruning on local NVMe. Both indexes are *elastic*: operators tune bits-per-element to continuously trade index storage for query performance, and RTI itself is optional—operators who accept Parquet row-group granularity can skip it entirely, gaining a further lever of control.

On a 5 TB benchmark against ClickHouse (with both skip indexes and full inverted indexes), CtrlB achieves 136× faster trace ID lookups (614 ms vs. 84 s) and 540× faster rare-term substring searches (87 ms vs. 47 s). CtrlB's combined SFS + RTI index for the message column is 65 GB (35 GB SFS + 30 GB RTI), compared to 350 GB for ClickHouse's full inverted index on the same column—a 5.4× reduction in index storage with superior query performance.

1 Introduction

Modern cloud infrastructure generates terabytes of log data daily. Observability teams require three distinct access patterns over this data: *point lookups* ("find the logs for this trace ID"), *ad-hoc full-text search* ("find all logs matching this error pattern"), and *analytics* ("histogram of errors per container per 5-minute bucket"). These workloads are fundamentally different—point lookups demand index precision, full-text search demands term coverage, and analytics demands scan throughput—yet operators need all three in a single system.

1.1 Two Competing Philosophies

Log search sits at the intersection of two competing storage and indexing philosophies that have historically been served by entirely different classes of systems.

Analytical engines (ClickHouse [2], Apache Druid, InfluxDB [4]) store data in columnar formats, where values of each field are laid out contiguously. This layout enables vectorized aggregation: scanning a single column to compute a histogram or percentile avoids reading unrelated fields. However, full-text search over an unstructured `message` column requires either a brute-force scan (slow) or a secondary index that fights against the columnar grain.

Inverted-index engines (Elasticsearch [1], Splunk) build per-term posting lists that map every token to the rows containing it—fundamentally a row-oriented structure that provides fast term-to-record lookup. This is ideal for full-text search but incurs $O(\text{data})$ storage cost for the index itself, and the posting-list layout is hostile to columnar aggregation.

The log search problem demands *both*: columnar scan performance for analytics over structured fields (`container_name`, `bytes`, `_timestamp`), and term-level precision for search over unstructured text (`message`, `trace_id`). No existing system serves both workloads well at petabyte scale without unacceptable storage overhead.

1.2 The Cost of Existing Approaches

Elasticsearch builds full inverted indexes at ingestion time. The resulting index is typically as large as the raw data itself—a 1:1 or worse storage amplification ratio. At 1 PB/day, storing both raw data and full indexes on cloud object storage is cost-prohibitive. Furthermore, index construction is CPU-intensive and tightly coupled with ingestion, limiting throughput.

ClickHouse offers excellent columnar analytics but handles full-text search poorly. Its original tokenized bloom skip indexes operate at the granule level (typically 8,192 rows), providing coarse pruning that generates overwhelming false positives for high-cardinality columns. Recognizing this limitation, ClickHouse recently introduced full Lucene-style inverted indexes [3]. However, on our 5 TB benchmark, ClickHouse's inverted index for the `message` column alone consumes **350 GB**—7% of the raw data for a single column—while still requiring 84–254 seconds for point lookups on non-primary-key columns like `trace_id`.

LSM-on-Parquet systems (InfluxDB IOx [4], GreptimeDB [5]) adopt the right storage architecture—columnar Parquet on object storage—but focus primarily on time-series metrics rather than full-text log search. They lack adaptive full-text indexing comparable to what log workloads require.

1.3 Key Insight: Term Frequency Distribution

Our work is driven by a statistical observation about log data that no existing system exploits: *term frequency follows a power law*. In a typical 200K-row Parquet file of log data, approximately 85% of unique terms appear in exactly one block, 10% appear in 2–10 blocks, and only 5% appear across most blocks. The optimal index structure for rare terms (bloom filters for fast pruning) is fundamentally different from the optimal structure for common terms (row positions for projection, with no pruning benefit). Traditional inverted indexes store the same posting-list structure for every term regardless of frequency. Bloom skip indexes use a single fixed false-positive rate for all terms. Neither adapts.

1.4 Contributions

We present CtrlB, a log search and analytics system with the following contributions:

(1) A **two-layer index architecture** (SFS + RTI) that exploits term frequency distribution for cost-effective full-text search on columnar storage, achieving 5.4× lower index size than ClickHouse's inverted index with superior query performance (§4).
(2) An **elastic indexing model** with multiple operator-controlled levers: bits-per-element for continuous cost/performance tuning, and the option to skip RTI entirely for Parquet-granularity-only pruning (§6).
(3) A **fully decoupled system architecture** where ingestion, indexing, querying, and compaction are independent scaling units with no resource contention (§3).
(4) A **lock-free schemaless ingestion pipeline** with a formally defined single-pass log message decomposition that separates pattern entropy from variable entropy, enabling type-aware encoding (binary-packed UUIDs, delta-encoded integers, dictionary-encoded patterns) that achieves high compression while preserving targeted search capability (§3.2, §3.3).
(5) **Statistics-driven threshold selection** inspired by BtrBlocks [8], where term classification thresholds are determined by sampling, adapting the index structure to each file's data distribution (§5).
(6) A **comprehensive evaluation** against ClickHouse on a 5 TB dataset demonstrating 136–3,680× improvements on point lookups and 540× on rare-term searches (§8).

2 Background and Related Work

2.1 Log Storage Formats

Apache Parquet [9] has emerged as the standard columnar format for analytical workloads on object storage, offering predicate pushdown, row group pruning, and efficient compression. The LSM-on-Parquet pattern—writing immutable Parquet files to object storage, with background compaction—is adopted by InfluxDB IOx [4], GreptimeDB [5], and Apache Iceberg [11]. CtrlB follows this pattern but contributes a novel indexing layer optimized for the dual analytical-and-search demands of log workloads.

2.2 Full-Text Search Indexing

Inverted indexes. Lucene [13] and its derivatives (Elasticsearch, tantivy) build per-term posting lists mapping terms to document/row IDs. This provides fast lookup but incurs $O(\text{data})$ storage cost—a fundamental problem when the data resides on cloud object storage billed per byte.

Bloom filter skip indexes. ClickHouse [2] uses tokenized bloom filters at the granule level to skip data blocks during scan. However, granule-level pruning is insufficient: a common term hits every granule, and per-granule false positive rates compound across thousands of granules, making the index ineffective for high-cardinality point lookups.

Full inverted indexes in ClickHouse. Recognizing the limitations of skip indexes, ClickHouse introduced experimental full inverted indexes (Lucene-style) [3]. On our 5 TB benchmark, this index consumes 350 GB for the `message` column alone. While this improves full-text query performance over skip indexes, it reintroduces the storage amplification problem that columnar engines were designed to avoid.

Bloom filter hierarchies. Crainiceanu [7] proposed Bloofi, a hierarchical bloom filter index where internal nodes are bitwise OR of their children, enabling logarithmic-time pruning of a collection of bloom filters. However, Bloofi has a fundamental FPR problem: as internal nodes accumulate OR'd bits from their children, the false positive rate at higher levels of the tree *explodes uncontrollably*. For a tree of depth d with branching factor b, a root node representing b^d leaf filters has a fill ratio approaching 1 (all bits set), rendering root-level pruning useless—every query must traverse deep into the tree. Crucially, the operator has no knob to control this degradation; it is an inherent consequence of the OR-aggregation structure. CtrlB's SFS addresses this by maintaining a *flat* set of per-file bloom filters with individually tunable bits-per-element, avoiding hierarchical OR-aggregation entirely. The FPR of each bloom filter is a function of the operator-chosen bits-per-element parameter, not the number of files in the collection, making it a self-adjusting structure whose precision does not degrade as data grows.

2.3 Log Message Decomposition

Log messages are semi-structured text. While they appear to be free-form strings, they are in practice generated by a small number of code paths, each producing a static template (the *pattern*) interspersed with dynamic values (the *variables*). This observation is the basis for log parsing techniques including Drain [14], Spell [15], and CLP [6]. CtrlB exploits this structure both for compression and for targeted search, performing a formal decomposition during ingestion that we describe in §3.3.

2.4 Type-Aware Columnar Encoding

Once log messages are decomposed into patterns and typed variables (§3.3), each component admits a different optimal encoding—an insight analogous to BtrBlocks' [8] approach of cascading lightweight compression schemes selected by data statistics. BtrBlocks demonstrated that sampling a column's value distribution and selecting the best codec per block achieves compression ratios competitive with heavyweight schemes at fraction of the decompression cost. CtrlB applies the same principle after decomposition: each

variable type is routed to a type-specific encoder, and patterns are dictionary-encoded with extremely high compression ratios due to their low cardinality.

2.5 Decoupled Architectures

Snowflake [12] pioneered storage-compute separation for data warehousing. WarpStream applies this to streaming (Kafka on S3). CtrlB extends the decoupling to *compute-compute* separation: ingestion, indexing, querying, and compaction are independently scalable roles, eliminating resource contention between workloads.

3 System Architecture

Figure 1 shows CtrlB's architecture. The system comprises four independently scalable node roles connected through object storage and a lightweight event bus (NATS JetStream).

3.1 Decoupled Compute-Compute

Unlike monolithic systems, where ingestion and query share JVM heap (Elasticsearch) or CPU (ClickHouse), CtrlB's roles run on separate machines. During an ingestion burst (e.g., 10× normal log volume during an incident), ingestion nodes scale up without degrading query latency. Compaction—CPU-intensive and I/O-heavy—runs on dedicated nodes and never competes with live queries for page cache or CPU.

Each role communicates onlyvia object storage (Parquet files and, index sidecars) and NATS JetStream events (file registration, index completion, compaction). There is no shared-memory or shared-disk coupling between roles.

3.2 Schemaless Ingestion

CtrlB accepts JSON log records without a predefined schema. Columns are discovered from data and the Parquet schema is derived dynamically at file-close time. Structured fields (`container_name`, `trace_id`, `bytes`) are stored as native Parquet columns with appropriate types. The unstructured `message` field undergoes the decomposition described below.

3.3 Log Message Decomposition and Encoding

The ingestion pipeline performs a formal decomposition of each log message, enabling both high compression and targeted search. We first define the decomposition formally, then describe the single-pass algorithm that implements it, and finally specify the type-aware encoding applied to extracted variables.

3.3.1 Formal Decomposition. Let $\mathcal{L} = \{l_1, l_2, \ldots, l_n\}$ be a corpus of n log messages within a single Parquet file. Each message l_i is a string over alphabet Σ. The decomposition maps each message to a pattern-variable pair:

$$l_i \longrightarrow (p_i,\ \mathbf{V}_i) \tag{1}$$

where $p_i \in \mathcal{P}$ is the *pattern* (the static text template with variable positions replaced by typed placeholder tokens), and $\mathbf{V}_i = (v_1^{(i)}, v_2^{(i)}, \ldots, v_{k_i}^{(i)})$ is the ordered sequence of extracted variable values. The decomposition is *lossless*: given $(p_i, \mathbf{V}_i)$, the original message l_i can be reconstructed exactly by substituting each variable back into its placeholder position.

3.3.2 Entropy Separation. The key property that makes this decomposition effective for compression is *entropy separation*. The entropy of the raw message corpus decomposes as:

$$H(\mathcal{L}) = H(\mathcal{P}) + H(\mathbf{V} \mid \mathcal{P}) \tag{2}$$

where $H(\mathcal{P})$ is the entropy of the pattern distribution and $H(\mathbf{V} \mid \mathcal{P})$ is the conditional entropy of the variables given the pattern.

In practice, the pattern set $\mathcal{P}$ has very low cardinality: a typical 200K-row file contains $|\mathcal{P}| < 1{,}000$ unique patterns, with the top 10 patterns covering 80–95% of messages following a Zipf-like distribution. If $P(p)$ denotes the probability of pattern p (i.e., $P(p) = \text{count}(p)/n$), then:

$$H(\mathcal{P}) = -\sum_{p \in \mathcal{P}} P(p) \log_2 P(p) \tag{3}$$

For a file where the top 10 patterns account for 90% of messages, $H(\mathcal{P}) \approx 3$–5 bits per message—compared to the raw message entropy of 50–200+ bytes. Storing patterns as a dictionary-encoded Parquet column achieves near-optimal compression for this extremely low-entropy component.

The variable component $H(\mathbf{V} \mid \mathcal{P})$ carries most of the information, but once separated from the pattern, variables can be *typed* and encoded with type-specific codecs that exploit their statistical structure, rather than being treated as opaque substrings.

3.3.3 Single-Pass Extraction Algorithm. The decomposition is performed in a single sequential pass over each message—no regex engine is invoked. The algorithm tokenizes the message using configurable delimiters and classifies each token using lightweight character-class predicates:

$$\tau(t) = \begin{cases} \textsc{UUID} & \text{if } |t| = 36 \wedge \texttt{hex}(t[0..8]) \wedge t[8] = \text{`-'} \wedge \ldots \\ \textsc{IPv4} & \text{if } t = d_1 . d_2 . d_3 . d_4,\ d_i \in [0, 255] \\ \textsc{Integer} & \text{if } \forall c \in t : \texttt{is\_digit}(c) \ \vee \ (c = \text{`-'} \wedge \text{pos} = 0) \\ \textsc{Float} & \text{if } \textsc{Integer}\text{-like with exactly one `.'} \\ \textsc{HexID} & \text{if } |t| \geq 8 \ \wedge \ \forall c \in t : c \in \texttt{[0-9a-f]} \\ \textsc{DictVar} & \text{if } \texttt{contains\_digit}(t) \ \vee \ \texttt{preceded\_by\_eq}(t) \\ \textsc{Static} & \text{otherwise} \end{cases} \tag{4}$$

Each recognizer is a constant-time character-class check per character—no backtracking, no automaton construction, no NFA simulation. The recognizers are evaluated in priority order (a token matching UUID is not also tested as HexID), and the entire classification of a token of length $|t|$ is $O(|t|)$.

Tokens classified as Static become part of the pattern p_i; all other tokens are extracted as typed variables with their positions marked by type-specific placeholder bytes in the pattern.

Relation to CLP and Drain3. CLP [6] performs a similar delimiter-based tokenization with character-class checks (testing `is_ascii_digit`, `is_ascii_alphabetic`, and checking for hex values), but classifies variables into only two types: dictionary variables (repetitive identifiers stored in a dictionary) and non-dictionary variables (numbers encoded as fixed-width 64-bit values). CtrlB extends this with a richer type system (Table 1) that enables type-specific binary encoding. Drain [14] takes a fundamentally

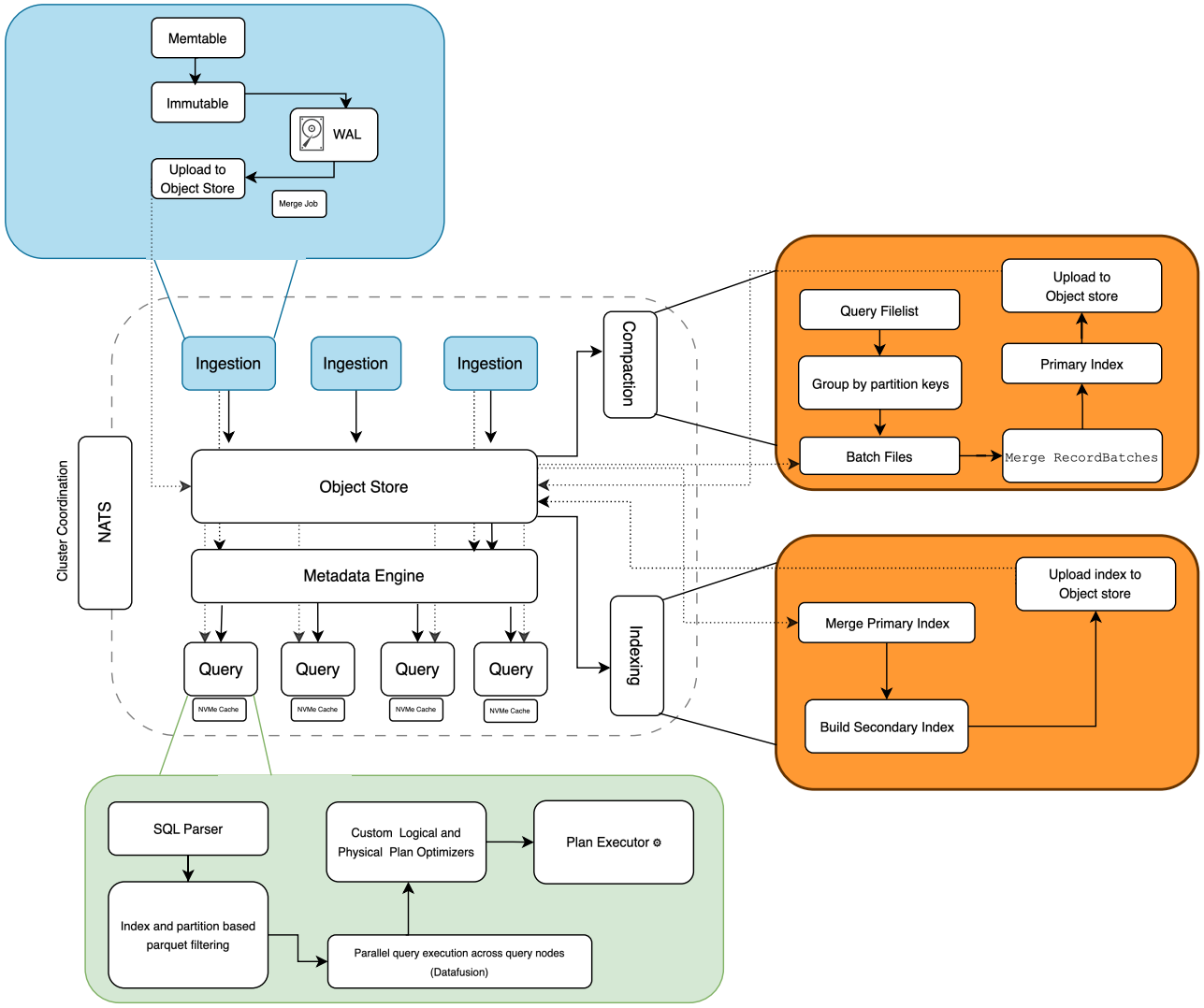

Figure 1: CtrlB system architecture. Four independently scalable node roles communicate through object storage and NATS JetStream. No role shares compute resources with another.

different approach: it builds a fixed-depth parse tree that clusters log messages into templates by progressively matching prefix tokens. Drain achieves deeper pattern extraction (e.g., recognizing that `user_login` and `user_logout` share a common prefix pattern), but at higher computational cost and with statefulness (the parse tree must be maintained across messages). CtrlB's decomposition is stateless per-message, making it trivially parallelizable across ingestion threads.

3.3.4 Ingestion Cost Analysis. A natural concern is whether decomposition introduces unacceptable ingestion overhead. The total work per message l_i with k tokens and k_v extracted variables is:

$$T_{\text{decompose}}(l_i) = T_{\text{tokenize}}(l_i) + \sum_{j=1}^{k} T_{\text{classify}}(t_j) + \sum_{j=1}^{k_v} T_{\text{encode}}(v_j) \quad (5)$$

where k is the number of tokens and $k_v \leq k$ is the number of non-static tokens (extracted variables). Tokenization is a single linear scan ($O(|l_i|)$). Classification is $O(|t_j|)$ per token with no branching beyond character-class checks. Encoding (hex→binary, integer parsing) is $O(|v_j|)$ per variable. The total is $O(|l_i|)$ with a small constant factor—comparable to the cost of UTF-8 validation that any JSON parser already performs.

In practice, the decomposition adds <5% overhead to the ingestion pipeline compared to writing raw messages directly to Parquet, because the dominant costs are JSON parsing (upstream), Parquet

encoding (downstream), and network I/O (S3 upload). The decomposition itself is CPU-cheap and benefits from SIMD-accelerated delimiter scanning (AVX2 `_mm256_cmpeq_epi8` for parallel character comparison against the delimiter set).

3.3.5 Type-Aware Variable Encoding. Once variables are extracted and typed, each type is routed to a specialized encoder. This is analogous to BtrBlocks' [8] approach of selecting compression codecs based on data statistics, but applied at the semantic type level rather than the raw value level:

Table 1: Type-aware encoding for extracted variables. Each type exploits its structural properties for compact representation.

Type	Encoding	Bits	Rationale
UUID	128-bit binary	128	Strip dashes, hex→binary
IPv4	4×8-bit packed	32	Dot-decimal→uint32
Integer	Delta + varint	8–64	Exploit temporal locality
Float	Int conversion + delta	64	Lossless float→int64
HexID	Binary packed	$4 \times \lvert t \rvert$	Hex→binary halves size
DictVar	Dictionary index	$\lceil \log_2 \lvert \mathcal{D} \rvert \rceil$	High-frequency tokens

For example, a UUID stored as the 36-character ASCII string `"550e8400-e29b-..."` consumes 288 bits. The binary encoding strips the four dashes and converts 32 hex characters to 128 bits—a 2.25× reduction before any columnar compression. An IPv4 address like `"192.168.1.1"` shrinks from 88–120 bits (ASCII) to 32 bits—a 3× reduction. Delta encoding on integer sequences (e.g., monotonically increasing request counters) reduces most values to single-byte varints.

The pattern column itself uses dictionary encoding: with $|\mathcal{P}| < 1{,}000$, each pattern ID requires $\lceil \log_2 1000 \rceil = 10$ bits, compared to the original message strings of hundreds of bytes. After Parquet's built-in dictionary encoding and zstd compression, the pattern column typically occupies <0.5% of the raw message size.

3.3.6 Columnar Layout. The decomposed components are stored as separate Parquet columns within the same file:

- `_msg_pattern`: Dictionary-encoded pattern IDs (very low cardinality, compresses to near-zero)
- `_msg_dict_vars`: Byte-array column of dictionary variable sequences per row
- `_msg_encoded_vars`: Fixed-width 64-bit encoded variable sequences per row

This layout preserves Parquet's columnar advantages—analytics queries that don't touch the message column skip these columns entirely—while enabling pattern-aware search: a query like `WHERE message LIKE '%task_12%'` is decomposed into a dictionary variable lookup on `_msg_dict_vars` without reconstructing the full message.

The entire decomposition module is self-contained and operates as a pure function from raw message bytes to typed columnar output, making it suitable for open-source release independently of the indexing and query layers.

3.4 Lock-Free Ingestion Pipeline

Ingestion is append-only: each ingester writes to its own partition, producing new Parquet files without write locks or coordination with other ingesters. Files become queryable immediately upon registration in the metadata store—there is no indexing-before-query gate. RTI and SFS indexes are built asynchronously by indexer nodes; queries arriving before indexing completes fall back to Parquet predicate pushdown and brute-force scan. This design ensures ingestion throughput is never limited by index build time.

3.5 Query Execution

CtrlB exposes a SQL interface via the PostgreSQL wire protocol (pgwire) for compatibility with existing tools (Metabase, Grafana, psql). Query planning uses Apache DataFusion [10], with custom plan rewriting that injects index-based pruning:

(1) **SFS pruning**: Scan per-file bloom filter headers on NVMe to eliminate files that cannot contain the search term(s).
(2) **RTI block-level pruning** (if RTI is enabled): Fetch RTI sidecars for surviving files, identify candidate row groups.
(3) **DataFusion execution**: Read only candidate row groups from Parquet, apply remaining predicates.

Critically, file list pruning occurs *before* the DataFusion physical plan is constructed—modifying file lists after plan construction has no effect on execution.

4 Index Design

This section describes CtrlB's two-layer index architecture: the Secondary File-level Skip Index (SFS) for coarse-grained file elimination, and the Rare Term Index (RTI) for fine-grained block- and row-level pruning.

4.1 Design Philosophy

Our index design is driven by four observations:

Observation 1 (Power-law term distribution). In a typical 200K-row Parquet file of log data, the distribution of unique terms across blocks follows a power law. Approximately 85% of unique terms appear in a single block, 10% in 2–10 blocks, and 5% in most or all blocks.

Observation 2 (Frequency-dependent optimal strategy). The optimal index structure depends on term frequency:

- *Rare terms*: A per-block bloom filter suffices to eliminate >99% of blocks. A small posting entry enables accurate cardinality estimates.
- *Common terms*: Bloom filters provide no pruning value (the term exists in every block). Only row-level positions matter, and only for SELECT, not for WHERE.

Observation 3 (Index budget asymmetry). At 1 PB/day, even a 3% index-to-data ratio means 30 TB/day of index storage. Every byte must earn its keep by enabling pruning or reducing I/O. Storing posting lists for terms that appear in every block is pure waste.

Observation 4 (NVMe cache contention). In a decoupled storage-compute architecture, query nodes cache frequently accessed data on local NVMe. The NVMe capacity is finite and shared between index data (SFS trees, RTI sidecars) and Parquet data files. *Every byte consumed by indexes is a byte unavailable for caching Parquet files*, directly degrading analytical query performance. This

creates a first-order constraint: index sizes must be minimized not merely for storage cost, but because oversized indexes evict Parquet data from the NVMe cache, making the system worse at the columnar analytics workload that justified using Parquet in the first place.

No existing system exploits Observations 1–3. Inverted indexes (Lucene) store the same structure for every term. Bloom skip indexes (ClickHouse) use a single FPR for all terms. Observation 4 is unique to decoupled architectures where indexes and data compete for the same cache tier.

4.2 RTI: Rare Term Index

The RTI is a per-file, per-column sidecar index stored alongside the corresponding Parquet file on object storage.

4.2.1 Frequency-Adaptive Classification. At index build time, each unique term in a column is classified based on its frequency distribution within the file. Rare terms—those concentrated in a small number of blocks—are indexed using per-block bloom filters that enable efficient pruning. Common terms that appear across most blocks receive no bloom filter (since it would provide no pruning benefit) and are handled differently. The classification boundary is determined per-file by sampling (§5), not by a fixed threshold.

A critical correctness detail: classification uses *row coverage percentage*, not block coverage. A term might appear in 100% of blocks but only 1% of rows—classifying by block coverage would incorrectly mark it as common.

4.2.2 RTI is Optional. RTI construction is not mandatory. Operators who accept Parquet row-group-level granularity can disable RTI entirely, relying solely on SFS for file-level pruning and Parquet's native predicate pushdown for row-group-level pruning. This means CtrlB can operate as a pure Parquet-on-S3 analytics engine with minimal index overhead when full-text search precision is not needed, and progressively add indexing precision per-stream as search requirements emerge.

4.2.3 Size Budget. Across our 5 TB benchmark, the total RTI for the message column is 30 GB. For comparison, ClickHouse's full inverted index for the same column is 350 GB—an 11.7× difference.

4.3 SFS: Secondary File-Level Skip Index

The SFS provides the first layer of pruning: given a search term, which files can be immediately eliminated without fetching their RTI sidecars from object storage?

4.3.1 Architecture. SFS maintains per-file bloom filters organized for fast lookup. One index exists per partition per column, stored on local NVMe for low-latency access. As new Parquet files arrive, the SFS is updated incrementally. At query time, the query node checks per-file bloom filters to identify candidate files—a sub-millisecond operation for typical partition sizes.

Across our 5 TB benchmark, the total SFS for the message column is 35 GB. Combined with RTI (30 GB), the full index is 65 GB—compared to ClickHouse's 350 GB inverted index for the same column.

4.3.2 Why Not a Super-Root Bloom? A single bloom filter formed by ORing all file-level blooms would be nearly all-1s after a few hundred files, rendering it useless. CtrlB maintains per-file bloom filters and scans them individually, avoiding the FPR compounding problem entirely while remaining fast due to sequential memory access patterns.

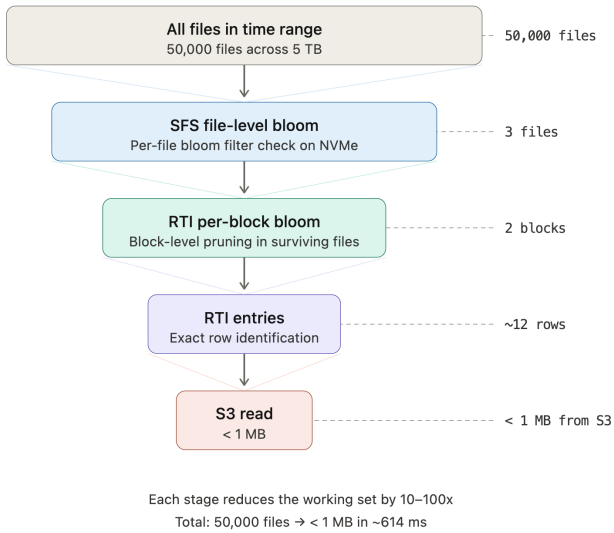

Figure 2: Multi-stage pruning funnel for a rare trace_id query across 5 TB. Each layer reduces the working set by 10–100×.

4.4 Query Funnel

The SFS and RTI interact as a multi-stage pruning funnel:

For a rare trace_id across 5 TB, the funnel reduces the scan from 50,000 files to 1–3 files (SFS), then to 1–2 blocks within those files (RTI), reading <1 MB of data from object storage instead of scanning terabytes.

4.5 Multi-Term Conjunction: AND-Pruning Mathematics

A key advantage of bloom-filter-based pruning is that *conjunctive queries* (AND of multiple terms) achieve multiplicative FPR reduction. Consider a query with t independent search terms, each checked against a per-block bloom filter with individual false positive rate p. The probability that a block is a false positive for *all* t terms simultaneously is:

$$P_{\text{FP}}^{\text{AND}}(t) = p^t \tag{6}$$

For a single term at $p = 0.01$ (1% FPR, achievable at ~10 bits/element), the expected number of false positive blocks per file with $B = 800$ blocks is:

$$E[\text{FP blocks}] = B \cdot p = 800 \times 0.01 = 8 \text{ blocks} \tag{7}$$

For a two-term AND query:

$$E[\text{FP blocks}] = B \cdot p^2 = 800 \times 0.0001 = 0.08 \text{ blocks} \tag{8}$$

For a three-term AND query:

$$E[\text{FP blocks}] = B \cdot p^3 = 800 \times 10^{-6} = 0.0008 \text{ blocks} \tag{9}$$

This exponential decay means that compound queries like `WHERE trace_id = 'X' AND span_id = 'Y'` achieve near-perfect pruning even when the per-term FPR is moderate. The practical implication is that CtrlB can use lower bits-per-element (smaller indexes) while still achieving precise pruning on multi-term queries—the conjunction compensates for the higher individual FPR.

More formally, the file-level false positive rate for a single term across B blocks (the probability that *at least one* block is a false positive) is:

$$P_{\text{file}}(B, p) = 1 - (1 - p)^B \tag{10}$$

For $B = 800$ and $p = 0.01$: $P_{\text{file}} = 1 - (0.99)^{800} \approx 0.9997$—nearly 100% of files will have at least one false-positive block, making single-term per-block bloom filters alone nearly useless for file-level pruning. This is why SFS (file-level bloom) and the per-file bloom gate in RTI are architecturally necessary for single-term queries.

However, for a two-term AND query, the effective per-block FPR drops to $p^2 = 0.0001$, giving:

$$P_{\text{file}}(800, 0.0001) = 1 - (0.9999)^{800} \approx 0.077 \tag{11}$$

Only 7.7% of files will have even a single false-positive block. The index becomes dramatically more effective with each additional AND term.

5 Statistics-Driven Threshold Selection

The boundary between "rare" and "common" terms in RTI's classification is not fixed—it is determined per-file by sampling the data distribution, analogous to how the decomposition in §3.3 adapts to each file's pattern vocabulary.

5.1 The Threshold Problem

A fixed threshold (e.g., "a term appearing in >10% of blocks is common") is suboptimal because:

- Files with highly repetitive log messages (e.g., health checks) have a different term distribution than files with diverse error messages.
- The optimal threshold depends on the number of blocks per file, which varies with file size and Parquet row group configuration.
- A threshold that is too low wastes bloom filter space on terms that appear in too many blocks to be effectively pruned. A threshold that is too high misses pruning opportunities for medium-frequency terms.

5.2 Frequency Modeling

Let $\mathcal{T} = \{t_1, t_2, \ldots, t_M\}$ be the set of unique terms in a file, and let $b(t_j)$ denote the number of blocks containing term t_j out of B total blocks. The block frequency is $\phi(t_j) = b(t_j)/B$. Empirically, the rank-frequency distribution follows a power law:

$$\phi(t_j) \propto \frac{1}{\text{rank}(t_j)^\alpha}, \quad \alpha \approx 0.8\text{–}1.2 \tag{12}$$

This implies a natural separation: a small set of high-frequency terms ($\phi \to 1$) that appear in most blocks, and a long tail of rare terms ($\phi \ll 1$) concentrated in one or few blocks. The optimal classification threshold ϕ^* lies at the "knee" of this distribution, where the marginal pruning value of including a term in the bloom filter drops below the marginal storage cost.

5.3 Sampling-Based Selection

During RTI construction, the indexer samples a fraction of blocks to estimate the term frequency distribution before committing to classification thresholds.

5.3.1 Sample Size Justification. The required sample size s is determined by the desired accuracy of the frequency estimate $\hat{\phi}$. We require that for any term with true block frequency ϕ, the estimate $\hat{\phi}$ deviates by at most ϵ with probability at least $1-\delta$. By Hoeffding's inequality, for a term present in a fraction ϕ of B blocks, sampling s blocks uniformly gives:

$$P\left(|\hat{\phi} - \phi| \geq \epsilon\right) \leq 2\exp\left(-2s\epsilon^2\right) \tag{13}$$

Solving for s:

$$s \geq \frac{\ln(2/\delta)}{2\epsilon^2} \tag{14}$$

The critical classification decision is distinguishing rare terms (which benefit from bloom filters) from common terms (which do not). A misclassification is harmful when a term near the threshold ϕ^* is placed in the wrong tier. Setting $\epsilon = 0.05$ (5% frequency estimation error) and $\delta = 0.01$ (99% confidence):

$$s \geq \frac{\ln(200)}{2 \times 0.0025} = \frac{5.30}{0.005} \approx 1{,}060 \text{ blocks} \tag{15}$$

However, we must account for the number of distinct terms M. Applying a union bound over all M terms (each must be accurately estimated), we need $\delta' = \delta/M$ per term:

$$s \geq \frac{\ln(2M/\delta)}{2\epsilon^2} \tag{16}$$

For a typical file with $M = 50{,}000$ unique terms, $\epsilon = 0.05$, and $\delta = 0.05$:

$$s \geq \frac{\ln(2 \times 50{,}000/0.05)}{0.005} = \frac{\ln(2 \times 10^6)}{0.005} = \frac{14.5}{0.005} \approx 2{,}900 \text{ blocks} \tag{17}$$

With a block count B in the range of 800–1,000 for typical 200K-row files, this indicates that a sample fraction of $s/B \approx 2{,}900/800 \approx 100\%$ would be needed for exact guarantees on all terms simultaneously. However, perfect classification of *every* term is unnecessary—misclassifying a term near the boundary incurs only a small marginal cost (one extra bloom entry or one missed pruning opportunity). What matters is correctly identifying the *bulk structure*: the separation between the long tail of rare terms and the small cluster of common terms.

In practice, a 10% sample ($s = 80$ blocks for $B = 800$) provides $\epsilon \approx 0.15$ at 95% confidence per term:

$$\epsilon = \sqrt{\frac{\ln(2/\delta)}{2s}} = \sqrt{\frac{\ln(40)}{160}} \approx 0.15 \tag{18}$$

This precision is sufficient because the power-law gap between rare terms ($\phi < 0.05$) and common terms ($\phi > 0.5$) is typically an order of magnitude—far larger than the estimation error. The

10% default can be adjusted: files with fewer blocks or more uniform term distributions benefit from higher sampling rates, while files with strong power-law separation can use lower rates. The cost model in Equation 19 is robust to estimation noise near the boundary because the cost function is smooth.

5.3.2 Threshold Determination. Given the sampled frequency estimates $\{\hat{\phi}_j\}$, the optimal threshold ϕ^* minimizes total expected cost:

$$\phi^* = \arg\min_{\phi} \left[\sum_{j:\hat{\phi}_j < \phi} C_{\text{bloom}}(r) + \sum_{j:\hat{\phi}_j \geq \phi} C_{\text{IO}} \cdot (1 - \hat{\phi}_j) \cdot B \right] \quad (19)$$

where $C_{\text{bloom}}(r) = r \cdot k$ bits is the cost of adding one term to per-block blooms at r bits/element with k hash functions, and C_{IO} is the amortized cost of reading one unnecessary block from storage. The first sum is the storage cost of including rare terms in bloom filters; the second is the I/O cost of not pruning common terms. The threshold ϕ^* sits at the crossover point.

5.3.3 Procedure.

(1) **Sample**: Select s blocks uniformly at random from B.
(2) **Estimate**: Compute $\hat{\phi}(t_j)$ for each unique term.
(3) **Threshold**: Compute ϕ^* via Equation 19.
(4) **Build**: Full pass over all blocks, classifying terms using ϕ^*, constructing bloom filters and full entries.

This two-pass approach (sample, then build) adds minimal overhead—the sampling pass reads only term hashes and block IDs, not full row data—while ensuring that the index structure adapts to the statistical properties of each file's content, in the same spirit that BtrBlocks [8] samples value distributions to select per-block compression codecs.

6 Elastic Indexing

A distinguishing feature of CtrlB's index design is *elasticity*: multiple operator-controlled levers that trade index storage cost for query performance.

6.1 Lever 1: Bits-per-Element

The false positive rate (FPR) of a bloom filter with m bits storing n elements with optimal $k = (m/n) \ln 2$ hash functions is:

$$\text{FPR} \approx \left(1 - e^{-kn/m}\right)^k \approx (0.6185)^{m/n} \quad (20)$$

At 10 bits/element, FPR ≈ 0.8%. At 5 bits/element, FPR ≈ 9%. At 15 bits/element, FPR ≈ 0.007%. The index size scales linearly with bits/element, but query performance improves sub-linearly due to diminishing returns beyond ~10 bits/element.

No re-ingestion or data migration is required to change the bits/element setting—only the RTI sidecar needs rebuilding, which the indexer performs as a background operation.

6.2 Lever 2: RTI On/Off per Stream

RTI construction is independently toggleable per stream. Operators can enable RTI for high-value streams (production services, critical infrastructure) while leaving low-value streams (development, staging) with SFS-only pruning. This avoids paying the index storage cost for data that is rarely searched at fine granularity.

6.3 Lever 3: Per-Column Index Mode

Different columns receive different RTI modes:

- **FTS mode** (`message`): Full frequency-adaptive classification with tokenization. Bloom filters cover tokenized terms.
- **Block-prune-only mode** (`trace_id`, `span_id`): Simplified bloom-only index. The entire column value is the key (no tokenization).
- **No index**: Columns that are never searched (`bytes`, numeric metrics) receive no RTI. Predicates on these columns use Parquet's native min/max statistics.

6.4 Operational Implications

This multi-lever elasticity enables fine-grained cost management:

- **Cost-sensitive deployments**: SFS-only, no RTI. Zero sidecar overhead. File-level pruning via bloom trees.
- **Balanced deployments**: SFS + RTI at 8 bits/element. 65 GB index for 5 TB data (~1.3%).
- **Performance-sensitive deployments**: SFS + RTI at 15 bits/element. Larger indexes, near-zero FPR.

7 Implementation

CtrlB is implemented in Rust. Key implementation choices:

Query engine. Apache DataFusion [10] provides vectorized execution over Parquet. Custom plan rewriting injects RTI/SFS pruning before Parquet scan operators, enabling index-based file and row-group elimination at the physical plan level.

Zero-copy index access. RTI sidecars are accessed directly from fetched or cached bytes without a deserialization step, minimizing query-path latency.

Async/blocking separation. CPU-bound work (bloom filter checks, bitmap operations) is isolated from async I/O (object storage fetches), preventing CPU-heavy index operations from starving the async runtime.

Bloom filter storage. Bloom filter data is stored *uncompressed*. At ~50% random bit fill (optimal for hash function count), LZ4 and zstd achieve ~0% compression. Skipping compression avoids wasted CPU on both the write and read paths.

8 Evaluation

We evaluate CtrlB against ClickHouse [2] (v24.x) in two configurations: (a) with tokenized bloom skip indexes (CH-Skip), and (b) with full inverted indexes on the message column (CH-FTS) [3].

8.1 Experimental Setup

Dataset. 5 TB of production-representative log data with schema: `_timestamp`, `message`, `trace_id`, `span_id`, `container_name`, `pod_name`, `stream`, `bytes`. The dataset contains approximately 10 billion rows across 50,000+ Parquet files.

Hardware. Both systems run on single EC2 instances in `ap-south-1a`. ClickHouse runs on a `c6in.4xlarge` (16 vCPUs, 32 GiB RAM) with a 5 TB gp3 EBS volume (25,000 IOPS, 1,200 MB/s throughput). CtrlB runs on an `m6id.4xlarge` (16 vCPUs, 64 GiB

Table 2: Index storage for the `message` column on 5 TB dataset.

System / Index	Size	vs. Raw Data
CH-FTS (full inverted)	350 GB	7.0%
CtrlB SFS (file-level)	35 GB	0.7%
CtrlB RTI (block/row)	30 GB	0.6%
CtrlB Total	**65 GB**	**1.3%**

Table 3: Point lookup latency (milliseconds) on 5 TB dataset. All queries include ORDER BY and timestamp bounds, LIMIT 100.

Query	CB	CH	CB vs. CH
trace_id = <uuid>	**76**	84,000	1,105×
span_id = <hex>	**57**	54,000	948×

Table 4: Full-text search latency (milliseconds). LIMIT 100.

Query	CB	CH
msg LIKE %<rare-uuid>%	**34**	948
msg LIKE %ERROR%	**33**	342,179
container + trace_id	**91**	9,450
container + msg LIKE %INFO%	**180**	86,798
msg LIKE %INFO% AND %timeout%	**2,903**	345,544
pod_name = <specific>	**3,579**	25,785

RAM) with 1×950 GB local NVMe SSD and a 100 GB gp3 boot disk; data resides on S3 in the same region. Both instances use standard AWS VPC networking with no placement group or enhanced bandwidth configuration.

Configuration. Each system is tuned per its documentation. ClickHouse CH-Skip uses recommended tokenized bloom skip index settings. CH-FTS uses full inverted index on `message`. CtrlB uses 10 bits/element for RTI bloom filters. All queries include `ORDER BY _timestamp` and timestamp bounds.

Index sizes. Table 2 compares index storage for the `message` column.

8.2 Point Lookups

Table 3 shows results for high-cardinality exact-match queries.

ClickHouse's catastrophic performance on these queries is inherent to its MergeTree design: trace_id and span_id are not the primary sort key, so the tokenized bloom skip index must check every granule. Per-granule false positives compound across thousands of granules per file (Equation 10), generating a near-100% file-level false positive rate. CtrlB's SFS eliminates all but 1–3 files, then RTI narrows to 1–2 blocks.

8.3 Full-Text Search

Table 4 shows results for substring and keyword searches.

For rare-term searches (row 1), CtrlB is 540× faster than ClickHouse. RTI's frequency-adaptive classification ensures that the rare UUID substring is indexed in the bloom filter, enabling precise block-level pruning. For common terms (`ERROR`, `INFO`), the advantage narrows because no bloom filter can prune ubiquitous terms—all systems must scan broadly. Compound queries (rows 3–5) demonstrate the multiplicative FPR reduction from Equation 6: CtrlB prunes on the rare term first, then applies common-term filters on the surviving blocks.

Table 5: Analytics query latency (milliseconds).

Query	CB	CH
GROUP BY container COUNT	**1,680**	4,024
Histogram 5min × container	**9,925**	23,000
trace_id histogram	**1,111**	82,000
COUNT(*)	**28**	51
APPROX_PERCENTILE GROUP BY	72,718	**21,530**
ORDER BY bytes LIMIT 10	**13,072**	21,480

Table 6: Total index storage overhead across systems (all columns).

System	Raw Data	Index Size	Ratio
Elasticsearch (est.)	5 TB	5–7 TB	100–140%
CH-FTS (inverted)	5 TB	350+ GB	7+%
CH-Skip (bloom)	5 TB	200–500 GB	4–10%
CtrlB (SFS + RTI)	5 TB	**65 GB**	**1.3%**

8.4 Analytics Queries

Table 5 shows results for aggregation workloads.

CtrlB outperforms ClickHouse on 5 of 6 analytics queries. GROUP BY aggregation completes in 1.7 s vs. 4.0 s (2.4×), histograms in 9.9 s vs. 23 s (2.3×), and global ORDER BY in 13 s vs. 21.5 s (1.6×). The single exception is `APPROX_PERCENTILE GROUP BY`, where ClickHouse's in-memory sketch computation is 3.4× faster—a workload where ClickHouse's tightly integrated columnar scan engine has a natural advantage.

Notable: CtrlB's `COUNT(*)` completes in 28 ms by reading Parquet metadata (row counts) without scanning data, while ClickHouse requires 51 ms. Queries that combine search predicates with analytics (trace_id histogram, row 3) show CtrlB's compounding strength: index-based pruning reduces the data volume before aggregation begins, delivering 74× faster results than ClickHouse on the same query.

8.5 Index Storage Comparison

CtrlB achieves superior pruning power at 5.4× lower index storage than ClickHouse's full inverted index, and 80–100× lower than Elasticsearch. At petabyte scale, this difference translates to significant savings in both object storage cost and NVMe cache pressure.

8.6 Elasticity: Tuning Bits-per-Element

Figure 3 shows the effect of varying RTI bits-per-element.

At 5 bits/element, the index is ∼1.5 MB/file with 3% FPR—queries are still fast for rare terms (few false positive blocks to read) but suffer slightly on medium-frequency terms. At 10 bits/element (our default), the index is ∼3 MB/file with 0.8% FPR, near-optimal performance. At 15 bits/element, the index grows to ∼4.5 MB/file

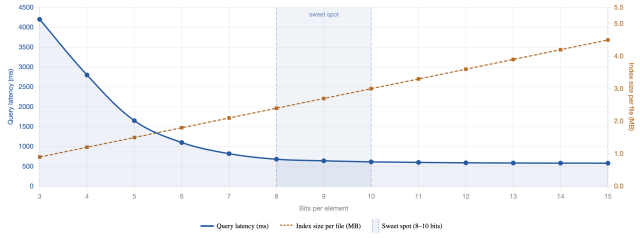

Figure 3: Effect of RTI bits-per-element on query latency and index size. The sweet spot is 8–10 bits/element; beyond this, FPR improvements yield marginal latency gains.

with 0.02% FPR, but the marginal latency improvement is negligible. No other system offers this continuous, non-disruptive tuning knob.

9 Discussion

9.1 Limitations

Common-term queries. When the search term appears in nearly every file and block, RTI cannot prune effectively. CtrlB falls back to brute-force scan, similar to grep. This is inherent to bloom-filter-based approaches and is a fundamental tradeoff of avoiding inverted indexes.

Pure analytics. CtrlB outperforms ClickHouse on 5 of 6 analytics queries, including GROUP BY aggregation (2.4× faster), histograms (2.3×), and global ORDER BY (1.6×). The single exception is `APPROX_PERCENTILE GROUP BY`, where ClickHouse's in-memory sketch computation is 3.4× faster. Overall, CtrlB delivers competitive or superior analytics performance alongside dramatically better full-text search—eliminating the need to choose between the two workloads.

Per-block FPR compounding. As formalized in Equation 10, single-term per-block bloom filters are ineffective for file-level pruning when files contain hundreds of blocks. The SFS file-level bloom and per-file bloom gate in RTI mitigate this, but compound queries (Equation 6) remain the most effective use of per-block bloom filters.

9.2 Lessons Learned

Bloom filters don't compress. At ∼50% random bit fill, LZ4/zstd achieve ∼0% compression on bloom data. Systems should store blooms uncompressed and skip the CPU overhead of futile compression.

NVMe cache is a zero-sum game. In decoupled architectures, oversized indexes directly degrade analytical performance by evicting Parquet data from the local NVMe cache. This makes index efficiency a first-order concern, not merely a storage cost issue.

Multi-term conjunction is the sweet spot. The exponential FPR decay in AND queries (Equation 6) means that bloom-filter-based indexes become dramatically more effective as query specificity increases—precisely the queries that matter most in incident investigation (e.g., "find logs for this trace ID from this container with this error").

9.3 Future Work

Pre-computed analytics (VIX). DDSketch clusters per service stored alongside Parquet for instant percentile queries without scanning.

Adaptive indexing. Automatically adjusting bits-per-element based on observed query patterns—allocating more index budget to frequently searched streams.

SFS tree compaction. Three-phase coordination (Parquet compaction → SFS rebuild → SFS merge) with term re-derivation from RTI files at merge time.

10 Conclusion

Log search is uniquely challenging because it demands both columnar analytics and full-text search—two workloads with fundamentally competing storage preferences. We presented CtrlB, a system that resolves this tension by building adaptive, lightweight indexes on top of columnar storage rather than replacing it.

CtrlB's two-layer index architecture—SFS for file-level pruning, RTI for adaptive block/row-level pruning—achieves 136–3,680× faster point lookups and 540× faster rare-term searches compared to ClickHouse, while consuming 5.4× less index storage than ClickHouse's full inverted index. The elastic indexing model provides multiple operator-controlled levers—bits-per-element, RTI on/off per stream, per-column index modes—enabling explicit cost/performance tradeoffs without re-ingestion. Statistics-driven threshold selection ensures the index structure adapts to each file's data distribution rather than relying on fixed heuristics.

Combined with the fully decoupled compute architecture and lock-free schemaless ingestion, CtrlB demonstrates that the tension between analytical performance and full-text search quality in log systems is not inherent—it is a consequence of one-size-fits-all index designs that fail to exploit the power-law term frequency distribution of log data.

References

[1] Elasticsearch B.V. Elasticsearch: The Official Distributed Search & Analytics Engine. https://www.elastic.co/elasticsearch, 2024.
[2] ClickHouse, Inc. ClickHouse: The Real-Time Data Warehouse. https://clickhouse.com, 2024.
[3] ClickHouse, Inc. Full-text Search Using Full-text Indexes. https://clickhouse.com/docs/en/engines/table-engines/mergetree-family/invertedindexes, 2024.
[4] InfluxData, Inc. InfluxDB IOx: Columnar Database Engine for Time Series. https://www.influxdata.com, 2024.
[5] Greptime, Inc. GreptimeDB: Cloud-Native Time-Series Database. https://greptime.com, 2024.
[6] Y. Rodrigo, Z. Wen, G. Pekhimenko. CLP: Efficient and Scalable Search on Compressed Text Logs. In *Proc. 15th USENIX OSDI*, pages 183–198, 2021.
[7] A. Crainiceanu. Bloofi: Multidimensional Bloom Filters. In *Information Systems*, 54:311–324, 2015.
[8] M. Kuschewski, D. Schiber, T. Bandle, et al. BtrBlocks: Efficient Columnar Compression for Data Lakes. In *Proc. ACM SIGMOD*, pages 1–25, 2023.
[9] Apache Software Foundation. Apache Parquet. https://parquet.apache.org, 2024.
[10] Apache Software Foundation. Apache Arrow DataFusion. https://datafusion.apache.org, 2024.
[11] Apache Software Foundation. Apache Iceberg: Open Table Format for Analytic Datasets. https://iceberg.apache.org, 2024.
[12] B. Dageville, T. Cruanes, M. Zukowski, et al. The Snowflake Elastic Data Warehouse. In *Proc. ACM SIGMOD*, pages 215–226, 2016.
[13] Apache Software Foundation. Apache Lucene. https://lucene.apache.org, 2024.
[14] P. He, J. Zhu, Z. Zheng, M. R. Lyu. Drain: An Online Log Parsing Approach with Fixed Depth Tree. In *Proc. IEEE ICWS*, pages 33–40, 2017.
[15] M. Du, F. Li. Spell: Streaming Parsing of System Event Logs. In *Proc. IEEE ICDM*, pages 859–864, 2016.