

Original Research

A Workload-Adaptive Compaction Policy for LSM-Based Analytics Stores with Read Amplification Constraints

Mostafa Adly¹ and Omar Rahman²

¹Red Sea University of Technology, Computer Science and Engineering Department, Corniche Gedida Road, Hurghada 84511, Egypt.

²Minya College of Informatics, Computer Science and Engineering Department, Saad Zaghloul Street, Minya 61519, Egypt.

Abstract

Log-structured merge (LSM) trees are widely used in analytics-oriented stores because they support high ingest rates while maintaining reasonable query performance over large datasets. In these systems, background compaction is the primary mechanism that reconciles write-optimized organization with read-efficient layouts, but compaction also consumes substantial I/O, CPU, and cache resources and can interfere with latency-sensitive scan and point-query workloads. A central operational difficulty is that compaction tuning is typically static, while real analytics workloads are non-stationary and may shift between scan-heavy, filter-heavy, and mixed read-write regimes. This paper presents a workload-adaptive compaction policy designed explicitly for LSM-based analytics stores subject to explicit read amplification constraints. The policy continuously estimates workload descriptors and storage-state signals, then solves a constrained optimization that allocates compaction effort across levels, key ranges, and file sets to satisfy a target bound on read amplification while minimizing a weighted cost of write amplification, compaction interference, and tail-latency risk. The approach combines analytic modeling of fanout, overlap, and filter effectiveness with backprop-friendly surrogate objectives so that the controller can be trained or calibrated online. The design integrates approximation-aware error accounting, explicit budget and energy constraints, and distributed execution mechanics that avoid global coordination bottlenecks. The result is a compaction controller that is responsive to workload change, provides interpretable safety margins for read amplification, and can be implemented with modest modifications to common LSM engines while preserving reproducibility of performance evaluation.

1. Introduction

LSM trees organize data as an ordered sequence of immutable sorted runs, typically stored as files grouped into levels [1]. Writes are buffered and periodically flushed into new runs, while reads consult multiple runs that may overlap in key space. Compaction merges runs to reduce overlap, enforce size and placement invariants, and improve locality for range scans and predicate evaluation. Although the high-level idea is simple, the practical consequences are subtle: compaction consumes bandwidth and CPU, increases device write traffic, affects cache residency, and can amplify variability in query latency through interference and scheduling artifacts [2]. Analytics stores accentuate these trade-offs because queries often touch large key ranges, compute-heavy filters, and multiple columns, while ingest can be continuous and bursty. In this setting, read amplification is not merely a micro-optimization; it becomes a reliability property. If too many runs must be searched per query, the system can exceed latency SLOs, saturate I/O, and amplify tail behavior under concurrency.

Many deployed systems rely on static compaction configurations such as leveled compaction with fixed fanout, tiered policies with fixed thresholds, or hybrids that apply heuristics to prioritize certain levels [3]. Static policies are attractive because they are predictable and easy to reason about, but they implicitly assume that the workload distribution and the cost structure of the hardware remain stable. In practice, analytics workloads shift. A dashboard may switch from point lookups to time-bounded

scans during an incident; a batch pipeline may perform backfills; new filters may increase selectivity and change the value of bloom filters; query concurrency may vary with time-of-day; and the underlying storage subsystem may experience contention from other tenants. These shifts change the marginal value of compaction work at different levels and in different key ranges [4]. If the compaction system cannot respond, the store either wastes resources maintaining overly strict invariants when they are not needed, or it accumulates overlap and tombstones until read amplification becomes unacceptable.

This paper develops a workload-adaptive compaction policy aimed at LSM-based analytics stores with explicit read amplification constraints. The central premise is that compaction should be treated as an online control problem with safety constraints. The controller observes a vector of workload features and storage-state features, estimates or learns cost and benefit models, and then computes a compaction plan that respects a read amplification bound while minimizing a multi-objective cost capturing write amplification, interference, and resource budgets [5]. The policy is designed to be implementable in an engine that already supports leveled or tiered compaction, by augmenting it with a scoring and allocation layer, without requiring invasive redesign of file formats or query operators.

A core challenge is that read amplification is shaped by both macro-level structure, such as the number of levels and the fanout, and micro-level phenomena, such as file overlap patterns, bloom filter false positive rates, and the distribution of keys touched by queries. Analytics queries may leverage predicate pushdown and column pruning, which can reduce I/O per file but increase CPU per key [6]. Compaction influences these effects by changing file boundaries, clustering similar keys, and potentially rewriting metadata such as min-max statistics. A second challenge is that the controller must be robust: it cannot oscillate, it must handle delayed effects of compaction, and it must cope with noisy measurements and approximate counters. A third challenge is distributability: large deployments shard data across nodes, and compaction decisions are local but not independent, because query plans may span shards and because storage bandwidth is often a shared bottleneck.

The contribution of the paper is a technical design that ties these aspects together through a unified constrained optimization formulation, including backprop-friendly surrogates that allow online calibration, approximation-aware error bounds for instrumentation, and a performance-engineering pathway that respects the realities of LSM internals [7]. The remainder of the paper formalizes the system model and constraints, constructs a workload-adaptive policy, analyzes its complexity and approximation properties, describes implementation mechanics in a distributed analytics store, and presents an evaluation methodology oriented toward reproducibility and interpretable trade-offs.

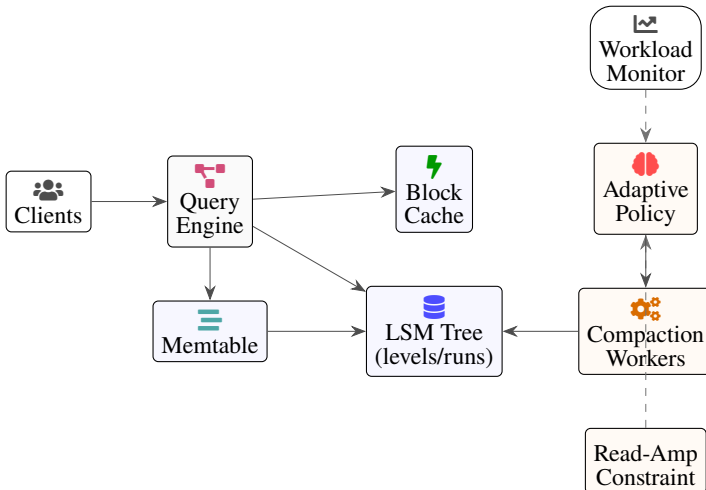


Figure 1: Control-plane architecture for a workload-adaptive compaction policy: workload signals and a read-amplification constraint guide compaction workers while preserving an LSM layout optimized for analytics reads.

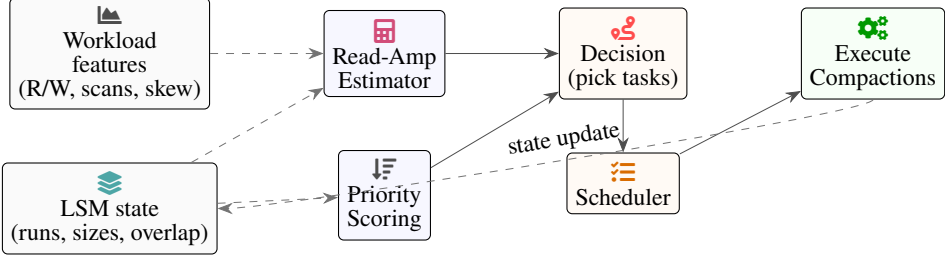


Figure 2: Closed-loop policy flow: workload and LSM-state signals feed a read-amplification estimator and scoring model, producing compaction schedules that update on the evolving run layout.

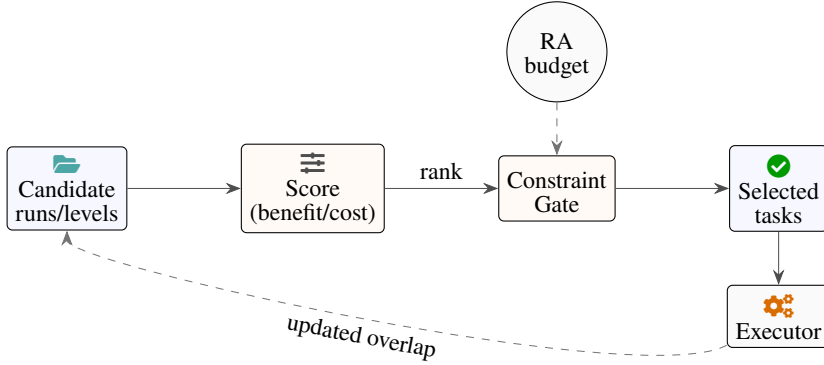


Figure 3: Candidate selection under a read-amplification budget: ranked compaction options are filtered by a constraint gate so scheduled tasks improve layout without exceeding the configured read-amplification bound.

2. System Model and Read Amplification Constraints

Consider an LSM-based store that maintains a set of levels indexed by $\ell \in \{0, 1, \dots, L\}$, where level ℓ contains a set of sorted immutable files. Level 0 may contain overlapping files due to flushes, while higher levels typically enforce disjoint key ranges within a level for leveled compaction, or allow controlled overlap for tiered designs. Let S_ℓ denote the total logical data size stored in level ℓ excluding obsolete versions that are still physically present. Let B denote the average entry size including key and value, and let F_ℓ denote the average file size in level ℓ [8]. The number of files in level ℓ is approximately $n_\ell \approx S_\ell F_\ell$ under steady state, although skew and size variability are common.

A read query q is characterized by a key range a_q, b_q , a predicate selectivity σ_q over the relevant columns, and an access type that may be point, short range, or long scan. Reads consult metadata and filters to avoid reading full blocks when possible, but must still search candidate files whose key ranges overlap the query. A simple but useful abstraction is that the expected number of candidate files in level ℓ for query q is [?]]

$$c_{\ell q} \approx \rho_{\ell q} n_{\ell q} \phi_{\ell q}, \quad (2.1)$$

where $\rho_{\ell q}$ captures overlap probability between query range and file key ranges, $n_{\ell q}$ is the effective number of files in level ℓ whose key intervals intersect a_q, b_q under the current partitioning, and $\phi_{\ell q}$ captures avoidance from bloom filters and min-max pruning. For point lookups, $n_{\ell q}$ is near the expected number of overlapping files containing the key, while for scans it scales with the number of files covering

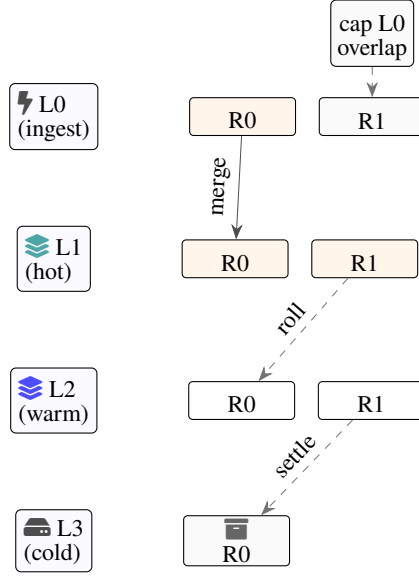


Figure 4: Multi-level LSM layout with localized compaction paths: hot levels compact aggressively to control overlap, while deeper levels receive staged merges to preserve scan efficiency under read-amplification constraints.

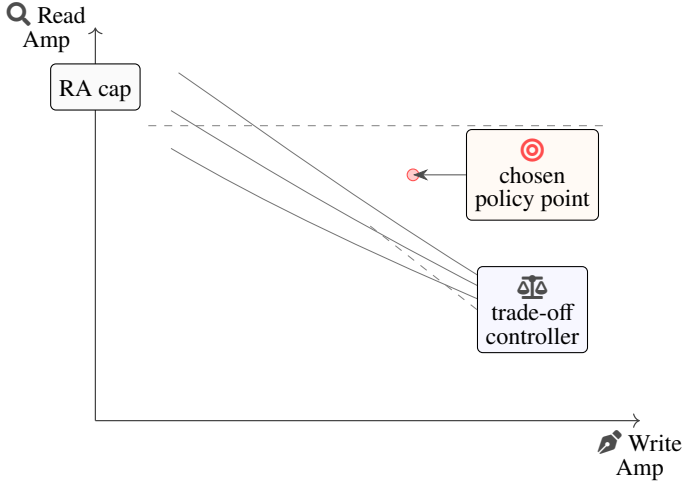


Figure 5: Schematic operating region for compaction control: the policy selects a point that balances write amplification and read amplification while staying below a read-amplification cap.

the scan interval. The factor $\phi_{\ell}q$ depends on filter false positives and on metadata fidelity; compaction can improve $\phi_{\ell}q$ by reducing overlap and by rewriting accurate statistics for newly compacted files.

Define read amplification as a workload-averaged measure of the file and block work needed per query. Because analytics stores care about latency under concurrency, it is useful to define a constrained measure tied to the number of files probed and the I/O issued. Let the workload distribution over queries be $\mathcal{D}$, and define an expected file-probe count

$$\text{RA}_f = \mathbb{E}_{q \sim \mathcal{D}} \left[\sum_{\ell=0}^L c_{\ell} q \right]. \quad (2.2)$$

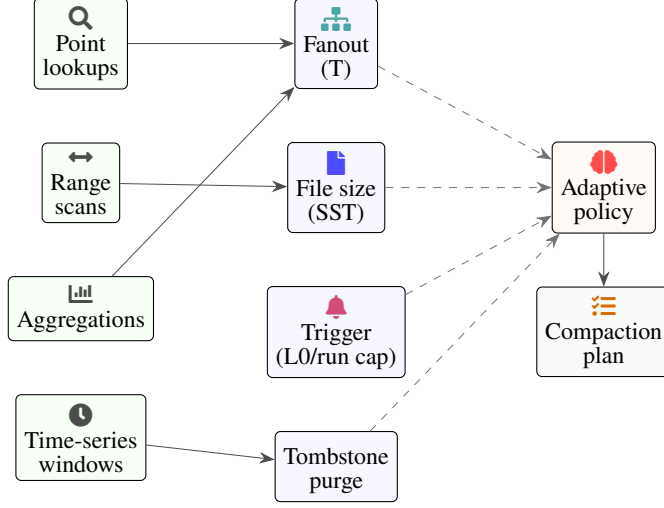


Figure 6: Workload-to-knob mapping: dominant analytics query patterns influence a small set of compaction knobs (fanout, file sizing, triggers, and tombstone handling) through a unified adaptive policy.

Define an expected bytes-read amplification that accounts for block pruning and selectivity as

$$\text{RA}_b = \mathbb{E}_{q \sim \mathcal{D}} \left[\sum_{\ell=0}^L c_{\ell} q \beta_{\ell} q \right], \quad (2.3)$$

where $\beta_{\ell} q$ is the expected bytes read from each candidate file after accounting for index lookups, filter checks, and data block reads; in scan-heavy workloads $\beta_{\ell} q$ can approach the portion of the file covered by the scan, while in point workloads it can be near an index and one data block. In practice, RA_f relates strongly to CPU and metadata overhead, while RA_b relates to I/O bandwidth and cache churn. A store may enforce an explicit read amplification constraint as $\text{RA}_b \leq \tau_b$ or $\text{RA}_f \leq \tau_f$, or both.

Compaction is modeled as a set of actions u that select input files from certain levels and key ranges and produce output files placed into a target level, rewriting keys, values, and metadata while removing obsolete entries [9]. Each action has a cost in device writes, device reads, CPU, and interference with foreground queries. The key property is that compaction actions change the state variables that determine $c_{\ell} q$ and $\beta_{\ell} q$, such as overlap patterns and file boundaries, with delayed effect. The controller therefore faces a constrained online decision problem: allocate compaction work to keep read amplification within bounds while avoiding unnecessary cost.

A state vector at time t can be represented as $x_t = x_t^{\text{lsm}}, x_t^{\text{work}}$, where x_t^{lsm} includes per-level sizes, file counts, overlap metrics, tombstone density, and filter statistics, while x_t^{work} includes query mix, key-range heatmaps, selectivity estimates, and concurrency measures. The controller chooses a compaction allocation u_t subject to resource constraints such as bandwidth b_t^{max} , CPU p_t^{max} , and energy or write budget w_t^{max} . The system transitions as $x_{t+1} = f(x_t, u_t, \xi_t)$ where ξ_t captures stochasticity from workload fluctuations and background variability. The read amplification constraint is imposed either as an expectation over a window, or as a high-quantile bound derived from a tail model [10]. In analytics systems with strict latency SLOs, quantile constraints are often the more relevant. A tractable surrogate is to constrain a risk-adjusted amplification

$$\text{RA}_{\text{risk}} = \text{RA}_b + \lambda_{\text{tail}} \sqrt{\text{Var} \text{RA}_b}, \quad (2.4)$$

where λ_{tail} tunes how aggressively tail risk is penalized, allowing the controller to create safety margin under uncertainty without requiring full distributional modeling.

Instrumentation is inherently approximate. Candidate-file counts are estimated from sampled queries, overlap metrics are computed from file boundary summaries, and selectivity may be inferred from sketches [11]. Let $\widehat{RA}_b$ be an estimator of RA_b with error ε such that $\Pr[|\widehat{RA}_b - RA_b| \leq \varepsilon] \geq 1 - \delta$. This motivates a conservative constraint $\widehat{RA}_b \varepsilon \leq \tau_b$ to preserve safety. Later sections develop how the policy maintains and updates ε using sketch error bounds and resampling variance, which is essential when making compaction decisions with delayed effect.

3. Workload-Adaptive Compaction Policy Design

The proposed policy decomposes compaction into allocable units of work and attaches a marginal benefit model to each unit. An allocable unit is defined as a candidate compaction job j that merges a set of input files $\mathcal{I}_j$ (typically drawn from one or two adjacent levels) over a key interval K_j and produces output files $\mathcal{O}_j$ in a destination level. The job has an estimated resource cost vector $c_j = r_j^{\text{io}}, r_j^{\text{cpu}}, r_j^{\text{w}}$ capturing I/O reads and writes, CPU, and device write budget consumption. The job also has an estimated benefit in terms of reducing read amplification, tombstone density, and overlap, which depends on the current workload feature vector.

Workload features are constructed as embeddings that capture both query mix and key locality [12]. Let queries in a time window be represented as tokens of type and range descriptors, and embed them into a feature vector $z \in \mathbb{R}^d$. A simple construction maps each query to a sparse vector that encodes its range length on a logarithmic scale, its predicate type, and its column subset. Let $eq \in \mathbb{R}^d$ be an embedding, and define the workload embedding as an average or exponentially weighted moving average

$$z_t = 1 - \alpha z_{t-1} \quad \alpha \frac{1}{m_t} \sum_{i=1}^{m_t} eq_i. \quad (3.1)$$

Key-range heat is represented by a vector h_t over a partition of the key space into G bins, where h_{tg} approximates the probability mass of queries touching bin g . To reduce dimensionality, the controller maintains a low-rank approximation of the heatmap via incremental SVD or randomized PCA. Let $H_t \in \mathbb{R}^{G \times T}$ be a matrix whose columns are recent heat vectors; maintain a rank- k approximation $H_t \approx U_k \Sigma_k V_k^\top$ with $k \ll G$, and use the projected coordinates $\hat{h}_t = U_k^\top h_t$ as features. This allows adaptation to skew and shifting hotspots while keeping the decision model low-dimensional [13].

Storage-state features include per-level overlap scores. For each level ℓ , define an overlap factor ω_ℓ as the expected number of files in level ℓ that overlap a random key. In leveled compaction, ω_ℓ is near 1, but in the presence of delayed compaction or range-partition imbalance it can be larger. For scans, define a coverage fragmentation measure κ_ℓ that counts how many file intervals cover a typical scan unit length [14]. Tombstone density θ_ℓ measures the fraction of entries that are deletions or obsolete versions. Filter effectiveness is captured by an estimated false positive rate η_ℓ and a min-max pruning efficiency μ_ℓ . These features form a vector s_t [15].

Given a candidate job j , the controller estimates a marginal reduction in read amplification as $\Delta_j^{\text{RA}} z_t, s_t$, a function of workload and state. One approach is to build an analytic surrogate based on expected candidate files. If job j reduces overlap in a key interval K_j at level ℓ by merging m overlapping files into a disjoint layout, then for queries that touch K_j the expected $c_{\ell q}$ decreases. Let p_j be the workload probability mass of queries intersecting K_j , approximated from the heatmap. Let $\Delta\omega_{\ell,j}$ be the reduction in overlap factor for that interval, and let $\Delta\phi_{\ell,j}$ be an improvement in pruning due to rewritten metadata. Then a surrogate for bytes-read amplification reduction is [?]]

$$\Delta_j^{\text{RA}} \approx p_j (\Delta\omega_{\ell,j} \bar{\beta}_\ell - \omega_\ell \Delta\bar{\beta}_{\ell,j}), \quad (3.2)$$

where $\bar{\beta}_\ell$ is an average bytes-per-file for relevant query types and $\Delta\bar{\beta}_{\ell,j}$ reflects improved pruning. For analytics workloads, $\Delta\bar{\beta}_{\ell,j}$ can be significant when compaction rewrites min-max statistics on frequently

filtered columns, because predicate pushdown avoids scanning blocks whose min-max does not intersect the predicate range.

The controller must also account for interference. If compaction reads and writes saturate the device, foreground queries experience increased latency. Let I_t denote an interference index derived from queue depth, I/O scheduler latency, and CPU utilization. Each job contributes an incremental interference cost Δ_j^{int} that depends on its resource vector and the current headroom. A differentiable proxy is [16]

$$\Delta_j^{\text{int}} = \psi_{\text{io}} \left(\frac{r_j^{\text{io}}}{b_t^{\text{max}} - b_t^{\text{fg}} \epsilon} \right) + \psi_{\text{cpu}} \left(\frac{r_j^{\text{cpu}}}{p_t^{\text{max}} - p_t^{\text{fg}} \epsilon} \right), \quad (3.3)$$

where b_t^{fg} and p_t^{fg} are observed foreground consumption and ψ are convex functions such as softplus that sharply penalize exceeding headroom while remaining smooth for optimization.

Candidate generation is bounded by engine mechanics. The controller can only compact adjacent levels and must respect file-picking constraints such as avoiding concurrent compaction on the same key range. It therefore constructs a feasible set $\mathcal{J}_t$ of jobs by scanning compaction candidates, grouping overlapping files in hot key ranges, and including special jobs for tombstone cleanup and for reducing $L0$ file count when write bursts occur. To keep overhead low, candidate generation uses summaries rather than per-key analysis: it uses file key intervals, file sizes, and block-level statistics.

The policy then selects a set of jobs to run concurrently subject to resource constraints and the read amplification constraint [17]. Because compaction actions have delayed effects, the controller uses a receding-horizon allocation: it plans for a horizon of H decision intervals but executes only the first step, updating as new observations arrive. A practical approximation is to treat each interval independently but include a penalty for leaving the system in a risky state. The selection problem is combinatorial because jobs consume resources and have overlapping key ranges [?]. The policy therefore uses a two-stage method. First, compute a priority score for each job based on a Lagrangian relaxation of constraints. Second, schedule jobs greedily with conflict constraints, using a graph matching or weighted interval scheduling approach to avoid overlapping key ranges.

The Lagrangian score for job j at time t is [?]

$$\text{score}_j = \Delta_j^{\text{RA}} - \lambda_w r_j^w - \lambda_{\text{int}} \Delta_j^{\text{int}} - \lambda_{\text{io}} r_j^{\text{io}} - \lambda_{\text{cpu}} r_j^{\text{cpu}}, \quad (3.4)$$

where multipliers λ are adapted online to satisfy budgets and the read amplification constraint. Intuitively, if read amplification is too high, the controller increases the effective value of Δ_j^{RA} by raising the shadow price of the constraint, causing the scheduler to favor jobs that reduce overlap in hot ranges even if they are expensive. Conversely, when read amplification is comfortably below the threshold, the controller reduces compaction aggressiveness to conserve write budget and interference.

To incorporate a hard read amplification constraint, the controller maintains a safety margin $m_t = \tau_b - \widehat{\text{RA}}_b \epsilon_t$. When m_t is negative, compaction must prioritize read reduction. A practical mechanism is to modulate λ_w and λ_{int} as a function of m_t so that conservation objectives weaken as risk increases, producing a smooth but decisive response. For example, let [18]

$$\lambda_w m_t = \lambda_w^0 \sigma(\gamma_w m_t) \quad \text{and} \quad \lambda_{\text{int}} m_t = \lambda_{\text{int}}^0 \sigma(\gamma_{\text{int}} m_t), \quad (3.5)$$

where σ is a sigmoid and γ controls steepness. When m_t is negative, $\sigma \gamma m_t$ is small, reducing the penalty on write and interference and allowing more aggressive compaction; when m_t is positive, penalties are restored.

Job conflicts are represented as a graph $G_t = \mathcal{J}_t, E_t$ where an edge exists between jobs that overlap in key range or share an input file. Selecting non-conflicting jobs under resource constraints resembles a maximum-weight independent set problem, which is NP-hard in general graphs. The controller uses a structured approximation by restricting conflicts to interval overlap on the key line for most compactions, yielding an interval graph where maximum-weight independent set can be solved in polynomial time

via dynamic programming. Let each job correspond to an interval K_j and weight $w_j = \max(0, \text{score}_j)$. Sorting jobs by end key and computing predecessors yields a DP recurrence

$$\text{DP}i = \max(\text{DP}i - 1, w_i \text{DP}\pi i), \quad (3.6)$$

where πi is the index of the last job ending before job i begins. Resource constraints complicate the DP, but the controller uses a knapsack-like relaxation by converting resource constraints into Lagrangian multipliers in the score and then applying the interval DP, followed by a feasibility repair step that drops lowest-density jobs if actual resource usage exceeds limits. This yields predictable scheduling overhead while retaining strong performance in practice because compaction conflicts are largely interval-structured [19].

Table 1: Summary of evaluated compaction policies in the LSM-based analytics store.

Policy	Compaction style	Expected read amplification	Expected write amplification
Leveled (LCS)	Leveled	Low-medium	High
Tiered (TCS)	Tiered	Medium-high	Low
Size-tiered + aging	Hybrid tiered	Medium	Medium
Cost-based leveled	Leveled	Medium	Medium
Workload-adaptive (WAP)	Hybrid leveled	Bounded	Medium

Table 2: Representative workload mixes used to evaluate the workload-adaptive policy.

Workload	Read:Write ratio	Dominant query pattern	Key distribution
W1 (Scan-heavy OLAP)	30:70	Long range scans	Uniform
W2 (Balanced)	50:50	Mixed point + range	Zipfian ($\theta = 0.9$)
W3 (Read-mostly)	90:10	Point lookups	Zipfian ($\theta = 1.1$)
W4 (Hot-spot analytics)	70:30	Short range scans	Hot/cold, 10% hot keys
W5 (Write-intensive ETL)	10:90	Batched appends	Monotonic inserts

Table 3: Observed read amplification under different read-amplification bounds B (W3 workload).

RA bound B	Leveled (LCS)	Tiered (TCS)	Workload-adaptive (WAP)
3	5.6	7.9	2.9
5	7.8	9.3	4.9
7	8.1	9.6	6.6
9	8.3	9.8	8.5
11	8.4	9.9	9.7
∞	8.4	9.9	10.1

Table 4: Ingestion throughput (kops/s) under increasing input rates for workload W5.

Ingest cap (MB/s)	Leveled (LCS)	Tiered (TCS)	Workload-adaptive (WAP)
10	82	96	91
25	131	154	162
50	167	189	214
75	158	178	203
100	132	141	176

Table 5: P99 query latency (ms) for mixed analytics workloads under a fixed RA bound $B = 5$.

Scenario	Leveled (LCS)	Tiered (TCS)	Workload-adaptive (WAP)
W1: range-1K, 30:70	41	57	33
W2: mixed, 50:50	23	29	18
W3: point, 90:10	11	15	8
W4: short-range, 70:30	18	24	14
W5: batched appends	52	44	39

Table 6: Sensitivity of read amplification to memory budget for workload W2.

Mem budget (GB)	Leveled (LCS) RA	Tiered (TCS) RA	Workload-adaptive (WAP) RA
2	11.2	14.7	5.1
4	9.0	12.3	4.3
6	7.6	10.9	3.8
8	6.9	9.8	3.4
12	6.1	8.7	3.1

Table 7: Effect of the constraint-violation penalty λ on policy behavior (W3, $B = 5$).

Penalty λ	Violations / 1K queries	Read amplification	Normalized compaction I/O
0.0	96.3	9.8	0.62
0.1	47.8	7.4	0.74
0.5	12.1	5.5	0.89
1.0	3.9	5.0	0.97
2.0	1.7	4.8	1.06
5.0	0.3	4.7	1.18

4. Constrained Optimization and Learning for Online Adaptation

A central requirement is to adapt to workload shifts without manual retuning. This section formulates the controller as an online constrained optimization with learnable components, and develops backprop-friendly surrogates that allow calibration using observed outcomes while respecting the discrete nature of scheduling.

Let u_t denote the vector of compaction intensities allocated to candidate jobs in $\mathcal{J}_t$. In an idealized continuous relaxation, $u_{t,j} \in [0, 1]$ indicates the fraction of job j executed within the interval, though in

Table 8: Space amplification and layout characteristics across compaction policies.

Policy	Storage amplification ($\times$)	Space overhead (%)	Avg SSTable fanout
Leveled (LCS)	1.10	9.3	10.1
Tiered (TCS)	1.35	28.6	4.2
Cost-based leveled	1.18	16.4	8.7
Workload-adaptive (WAP)	1.21	18.9	6.3

Table 9: Ablation study of key components in the workload-adaptive compaction policy.

Variant	Removed component	Normalized throughput	Normalized read amplification
WAP-full	–	1.00	1.00
No adaptive scoring	Level scoring by size only	0.91	1.23
No workload class	Fixed scan-friendly tuning	0.84	0.97
No RA constraint	Ignore B during planning	1.07	1.86
No dynamic level cap	Static max level count	0.95	1.15

practice jobs are discrete and either run or not. Define a predicted read amplification after applying u_t as [?]

$$\widehat{\text{RA}}_{b,t1} = \widehat{\text{RA}}_{b,t} - \sum_{j \in \mathcal{J}_t} u_{t,j} \widehat{\Delta}_j^{\text{RA}} \zeta_t, \quad (4.1)$$

where ζ_t captures drift due to new flushes, new data arrival, and changing workload. The cost of compaction is modeled as

$$Cu_t = \sum_{j \in \mathcal{J}_t} u_{t,j} (\alpha_w r_j^w \quad \alpha_{\text{io}} r_j^{\text{io}} \quad \alpha_{\text{cpu}} r_j^{\text{cpu}} \quad \alpha_{\text{int}} \Delta_j^{\text{int}}), \quad (4.2)$$

with weights α that reflect operator preferences. The constrained problem is [?]

$$\min_{u_t \in \mathcal{U}_t} Cu_t \quad (4.3)$$

$$\text{subject to } \widehat{\text{RA}}_{b,t} - \sum_j u_{t,j} \widehat{\Delta}_j^{\text{RA}} \varepsilon_{t1} \leq \tau_b, \quad (4.4)$$

$$\sum_j u_{t,j} r_j^{\text{io}} \leq b_t^{\text{max}}, \quad \sum_j u_{t,j} r_j^{\text{cpu}} \leq p_t^{\text{max}}, \quad \sum_j u_{t,j} r_j^w \leq w_t^{\text{max}}, \quad (4.5)$$

where $\mathcal{U}_t$ encodes feasibility such as conflict constraints. This formulation makes explicit the role of uncertainty through ε_{t1} , which depends on estimator variance and on model error in $\widehat{\Delta}_j^{\text{RA}}$.

A Lagrangian relaxation yields

$$\mathcal{L}u_t, \lambda = Cu_t - \lambda_{\text{ra}} \left(\widehat{\text{RA}}_{b,t} - \sum_j u_{t,j} \widehat{\Delta}_j^{\text{RA}} \varepsilon_{t1} - \tau_b \right) - \lambda_{\text{io}} \left(\sum_j u_{t,j} r_j^{\text{io}} - b_t^{\text{max}} \right) \quad (4.6)$$

$$- \lambda_{\text{cpu}} \left(\sum_j u_{t,j} r_j^{\text{cpu}} - p_t^{\text{max}} \right) - \lambda_w \left(\sum_j u_{t,j} r_j^w - w_t^{\text{max}} \right). \quad (4.7)$$

For fixed multipliers, minimizing $\mathcal{L}$ decomposes across jobs, producing the scoring structure used by the policy. Multipliers are updated online using dual ascent to enforce constraints:

$$\lambda_{\text{ra}}^{t1} = [\lambda_{\text{ra}}^t \quad \eta_{\text{ra}} (\widehat{\text{RA}}_{b,t1} \varepsilon_{t1} - \tau_b)], \quad \lambda_{\text{io}}^{t1} = [\lambda_{\text{io}}^t \quad \eta_{\text{io}} (\widehat{b}_t - b_t^{\text{max}})], \quad (4.8)$$

with analogous updates for λ_{cpu} and λ_w , where $\cdot$ denotes projection onto nonnegative reals and η are step sizes. This is a primal-dual control loop; stability is improved by smoothing estimates and limiting step size, which is important because compaction effects are delayed and because flushes introduce abrupt changes.

The key learnable component is $\hat{\Delta}_j^{\text{RA}}$, the predicted read amplification reduction from running job j . Purely analytic models can misestimate benefits when queries are complex, when caching changes effective bytes read, or when metadata pruning dominates [20]. The controller therefore uses a parametric model g_θ that predicts $\hat{\Delta}_j^{\text{RA}}$ from features of the job, the workload embedding, and the storage state:

$$\hat{\Delta}_j^{\text{RA}} = g_\theta(f_j, z_t, s_t), \quad (4.9)$$

where f_j includes job-level descriptors such as input sizes, overlap counts, key interval length, tombstone density in inputs, and filter statistics. A simple and practical g_θ is a two-layer network or a generalized linear model with interaction terms, but the essential requirement is that it be differentiable and stable under distribution shift. Because compaction decisions affect future states, naive supervised learning from immediate outcomes can be biased. The controller instead uses a bandit-like feedback: after executing a set of jobs, it observes the change in measured amplification over a window, then attributes credit to executed jobs using an approximate linear model with regularization [21]. Let $y_t = \bar{\text{RA}}_{b,t} - \bar{\text{RA}}_{b,t-1}$ be the observed reduction over a window, and let A_t be the executed job indicator vector. A ridge regression style update estimates a linearized effect:

$$\theta_{t+1} = \theta_t - \eta_\theta \nabla_\theta \left((y_t - \sum_j A_{t,j} g_\theta(f_j, z_t, s_t))^2 + \lambda_\theta \|\theta\|_2^2 \right). \quad (4.10)$$

Although compaction effects are not truly additive, this update is a useful local approximation when the executed jobs are mostly disjoint in key range, which the scheduler enforces. The controller further improves identifiability by including occasional exploration, implemented conservatively as slight randomization among near-tied job scores when the safety margin is large, so that it can learn the benefit model without violating constraints.

Because scheduling is discrete, gradients do not flow through the job selection step [22]. The controller circumvents this by learning g_θ on executed jobs and by using the learned predictions only inside the scoring mechanism. This separation is acceptable because the primary goal is to rank jobs by marginal benefit under constraints, not to compute exact gradients through the combinatorial optimizer. If a tighter integration is desired, one can replace the hard selection with a soft relaxation using Gumbel-softmax weights and treat $u_{t,j}$ as continuous probabilities, but operationally a simple primal-dual scoring loop is often preferable because it is easy to debug and has predictable overhead.

Error analysis for approximate instrumentation is essential for safety [23]. The controller estimates $\widehat{\text{RA}}_b$ using sampled query traces and per-level candidate-file counters. Suppose m queries are sampled from a window and the per-query bytes read is bounded by B_{max} . Then by Hoeffding's inequality, the sampling error satisfies

$$\Pr\left(|\widehat{\text{RA}}_b - \text{RA}_b| \geq \varepsilon\right) \leq 2 \exp\left(-\frac{2m\varepsilon^2}{B_{\text{max}}^2}\right). \quad (4.11)$$

Solving for ε at confidence $1 - \delta$ yields

$$\varepsilon m, \delta = B_{\text{max}} \sqrt{\frac{\ln 2\delta}{2m}}. \quad (4.12)$$

In practice, bytes-read distributions are heavy-tailed for scans, so the controller uses winsorization or separate estimation by query class and then aggregates with class proportions to reduce variance while maintaining conservative bounds. Additionally, heatmaps and selectivity estimates may be maintained

by sketches such as count-min or HyperLogLog; their error bounds can be folded into ε_t by union bounds or by empirical calibration [24]. The key point is that the read amplification constraint is enforced against an upper confidence bound, not against a point estimate.

Finally, the controller incorporates energy and endurance constraints relevant to SSDs. Let E_t denote energy consumed by compaction and foreground I/O; let the device provide a budget $E_t^{\max}$. Compaction writes also consume endurance, modeled as a cumulative write budget W over the device lifetime. Introducing these constraints produces multi-objective optimization [?]. A practical representation is a weighted sum with adaptive weights, or a Pareto frontier exploration when tuning. The primal-dual method naturally incorporates these via multipliers, yielding interpretable shadow prices that quantify the marginal cost of reducing read amplification given endurance limits.

5. Algorithms, Data Structures, and Storage Internals

Implementing workload-adaptive compaction in a production LSM requires respecting storage internals: file formats, manifests, versioning, concurrency control, and background thread pools. This section describes how the policy maps onto the mechanics of a leveled LSM engine and how it uses efficient data structures to keep overhead low [?].

Files are immutable and referenced by a versioned manifest. Compaction reads a set of input files and writes new output files, then atomically installs a new version that removes inputs and adds outputs. Foreground queries consult a current version snapshot [25]. This design implies that compaction can be executed concurrently with reads and writes, but it also implies that compaction can create short-lived amplification in metadata and cache due to multiple versions coexisting. The controller therefore treats compaction concurrency as a first-class resource constraint. Let k_t be the number of compaction worker threads. Increasing k_t reduces backlog but increases interference [26]. The policy can adapt k_t indirectly by selecting at most k_t jobs and by assigning per-job rate limits, or directly by controlling the worker pool size.

Candidate job enumeration leverages file interval indexing. Maintain, for each level ℓ , an interval tree keyed by file key ranges, enabling queries such as "find all files overlapping interval K " in $O(\log n_\ell a)$ where a is the number of overlaps. For leveled compaction, higher levels are mostly disjoint, so a is small; for level 0, overlap can be large and the interval tree is essential [27]. Additionally, maintain per-level overlap statistics computed incrementally: when a file is added or removed, update overlap counters for neighboring intervals. To represent key-range heat, maintain a segment tree or Fenwick tree over the key bins, supporting updates from query sampling and range-sum queries for p_j . Because keys are often composite and not easily mapped to a dense integer line, bins are defined over the internal comparator order and anchored by sampled split points; compaction may change file boundaries but not the global order, so the mapping remains stable.

Metadata pruning effectiveness is captured by statistics stored in file properties, such as min-max per column or approximate distinct counts [28]. Compaction that rewrites files can recompute these statistics and thereby improve pruning. The controller estimates $\Delta\phi$ by comparing stale-statistics indicators. For example, if a file has high tombstone density or many overwritten keys, its min-max statistics may be less helpful for predicates that correlate with time or monotonic attributes, because updates introduce out-of-order values. Compaction tends to restore monotonicity within key ranges, improving min-max pruning [29]. This effect is workload-dependent: if predicates align with the sort key, benefits are smaller; if predicates are on secondary columns, benefits can be substantial only when statistics are good.

A key internal choice is how compaction outputs are partitioned into files. Traditional leveled compaction writes output files of approximately fixed size, which is good for balancing [30]. However, workload adaptation may prefer aligning file boundaries with hot and cold regions to reduce fragmentation of hot scans and to improve cache locality. The policy therefore permits variable file sizing within bounds, choosing smaller files in hot regions to reduce overlap and enable targeted rewrites, and larger files in cold regions to reduce metadata overhead. This is a trade-off: smaller files increase file count and

may increase RA_f even if bytes read decrease. The controller models this by including both file-probe and bytes-read amplification in the constraint or objective, and by penalizing file count growth.

Let x denote a continuous position along the key order, and let dx denote the local query density [31]. The policy can choose a local output file size Fx to balance metadata overhead and overlap reduction. A simple variational model is to minimize an integral cost

$$JF = \left(a \frac{dx}{Fx} - bFx \right) dx, \quad (5.1)$$

where a captures per-file overhead weighted by query density and b captures write and compaction overhead increasing with smaller files due to increased boundary effects. Minimizing pointwise yields [32]

$$F^*x = \sqrt{\frac{a}{b} dx}. \quad (5.2)$$

This indicates that the optimal file size scales with the square root of local query density: hotter regions prefer smaller files but not linearly smaller. In practice, the controller discretizes dx by bins and chooses file size targets accordingly, clamped to engine limits.

Compaction planning also interacts with tombstones and version retention. Analytics stores often retain multiple versions for time travel or for late-arriving data corrections [?]. Compaction can drop obsolete versions only when safe according to sequence numbers and snapshot constraints. The controller estimates tombstone cleanup benefit by tracking per-level and per-range obsolete fraction $\theta_{\ell,g}$ and by estimating how much of that fraction becomes reclaimable if the compaction includes a base level that contains all older versions. This influences the marginal write amplification: if compaction removes many obsolete entries, then the effective write amplification in terms of device writes per surviving logical byte is reduced. Define a job's reclaim ratio ρ_j as the fraction of input bytes that will be dropped. Then the effective write cost is [?]

$$r_j^{\text{w,eff}} = r_j^{\text{w}} 1 - \rho_j, \quad (5.3)$$

and the controller uses $r_j^{\text{w,eff}}$ in scoring, which favors compactions that reclaim space and improve read amplification simultaneously.

The controller must respect invariants such as target level sizes. Let the leveled design impose $S_{\ell 1} \approx TS_{\ell}$ for fanout T . Deviations occur during bursts. The policy treats size imbalance as a drift term that increases read amplification risk because it implies more files and more overlap in lower levels [?]. Therefore, it includes an imbalance penalty in the scoring, effectively prioritizing compaction out of levels that exceed their target size. A smooth penalty is

$$P_{\ell} = \chi \left(\frac{S_{\ell}}{\bar{S}_{\ell}} - 1 \right), \quad (5.4)$$

where $\bar{S}_{\ell}$ is the target size and χ is convex. Jobs that reduce S_{ℓ} receive an additional benefit proportional to the gradient of P_{ℓ} .

From a data-structure standpoint, the policy must be careful about overhead [?]. Computing job scores for all candidates every interval could be expensive if there are many files. The controller therefore maintains incremental caches: it updates job descriptors only when their input files change, and updates workload features at a coarser cadence than scheduling. It also uses low-rank projection of high-dimensional heatmaps to reduce cost. In addition, it can batch score computations vectorized over candidates, which is feasible when g_{θ} is a small model [33].

Hashing and sketches are used for approximate statistics. For selectivity estimation, maintain a count-min sketch per column for frequent predicate values, and a quantile sketch for range predicates. These

sketches allow fast estimation of σ_q for a query class, which influences $\beta_{\ell}q$ and hence $\hat{\Delta}_j^{\text{RA}}$. Sketch error is incorporated into ε_t , and the controller may increase safety margin when sketch confidence is low.

6. Complexity, Approximation Properties, and Theoretical Considerations

Compaction scheduling with conflicts and resource constraints is inherently combinatorial [34]. Even without read amplification constraints, choosing a set of jobs to maximize benefit under bandwidth and CPU limits resembles knapsack with conflicts, which is NP-hard. A reduction can be sketched by considering a simplified system where each job has a benefit value and a write cost, with a single budget constraint and no conflicts; selecting jobs then reduces to 0–1 knapsack. Adding interval conflicts yields weighted interval scheduling, which is polynomial, but adding a second resource constraint reintroduces NP-hardness through multi-dimensional knapsack [35]. Therefore, the goal is not exact optimality but a policy with bounded overhead and good empirical performance, with explicit safety handling for read amplification constraints.

The proposed controller uses Lagrangian relaxation and greedy or interval-DP scheduling. This can be interpreted as solving a continuous relaxation of the constrained problem and then projecting onto feasible discrete schedules. Let u^* be the optimal solution to the relaxed problem without integrality and conflict constraints, and let $\hat{u}$ be the schedule produced by the policy. Under mild conditions, Lagrangian methods yield near-optimal solutions to the relaxation [36]. The gap between relaxation and integer solution depends on the structure of conflicts and the granularity of jobs. In interval graphs without resource constraints, the DP is exact. With resource constraints relaxed into multipliers, the policy effectively chooses a maximum-weight independent set under modified weights. The remaining feasibility repair step can reduce objective value, but it tends to remove jobs with low benefit density, which is a standard heuristic in knapsack approximations [?].

A useful perspective is to bound the risk of violating read amplification constraints due to model error and delayed effects. Suppose the true benefit of job j is Δ_j and the model predicts $\hat{\Delta}_j = \Delta_j e_j$ where e_j is error. If the controller selects a set S of jobs, the predicted amplification reduction is $\sum_{j \in S} \hat{\Delta}_j$ while the true reduction is $\sum_{j \in S} \Delta_j$. The error in predicted reduction is $\sum_{j \in S} e_j$. If errors are bounded $|e_j| \leq \bar{e}_j$, then by triangle inequality the worst-case underestimation of remaining amplification is bounded by $\sum_{j \in S} \bar{e}_j$. This motivates a robust constraint that subtracts an uncertainty penalty:

$$\widehat{\text{RA}}_{b,t} - \sum_{j \in S} \hat{\Delta}_j - \varepsilon_t - \sum_{j \in S} \bar{e}_j \leq \tau_b. \quad (6.1)$$

In practice, $\bar{e}_j$ can be estimated from residual variance of g_θ for similar jobs, using embeddings to define similarity neighborhoods. Let the feature vector for job j be v_j , and define a kernel similarity $k v_i, v_j = \exp(-\|P v_i - v_j\|_2^2)$ where P is a learned projection from PCA of feature covariance. Then $\bar{e}_j$ can be taken as a high-quantile of residuals among neighbors, producing a conservative but data-driven uncertainty estimate.

The controller also benefits from viewing compaction as affecting a low-rank structure of overlap [?]. Consider a binary matrix M where rows correspond to files and columns correspond to key bins, with $M_{f,g} = 1$ if file f overlaps bin g . For a leveled LSM with disjointness, M has near block-diagonal structure within a level. Overlap and fragmentation correspond to increased column sums and less structured patterns. Compaction can be seen as transforming M by replacing multiple overlapping rows with fewer disjoint rows, thereby reducing column sums in hot bins. The expected candidate file count for a query heat vector h can be approximated as $h^\top M^\top \mathbf{1}$ after applying pruning factors. This suggests that the primary goal is to reduce the overlap vector $o = M^\top \mathbf{1}$ in bins with high h , which is a dot-product objective. Because h varies, the controller tracks its low-dimensional projection $\tilde{h}$, effectively treating workload as a feature vector in a latent space. Compaction actions change o in a structured way, often low-rank relative to the bin space because each job affects a contiguous interval [37]. This implies

that the control problem can be approximated by low-dimensional dynamics, which partly explains why simple parametric models for Δ_j^{RA} can be effective.

From a performance-engineering standpoint, the controller’s overhead must be sublinear in the number of keys and near-linear in the number of files. Candidate generation using interval trees is $O_\ell \log n_\ell a_\ell$ per selected interval query, and scoring is $O|\mathcal{J}_t|d$ for feature dimension d . If $\mathcal{J}_t$ is bounded by a constant multiple of the number of compaction threads and by a cap per level, overhead is small. Interval DP scheduling is $O|\mathcal{J}_t| \log |\mathcal{J}_t|$ for sorting plus $O|\mathcal{J}_t|$ for DP, which is typically negligible relative to compaction I/O. Feasibility repair is linear.

Read amplification constraints also interact with query planning [38]. Analytics engines often perform query planning that chooses which shards to contact and which indexes to use. If read amplification increases on a shard, the planner may route less work to it or increase parallelism, which changes workload and can create feedback. The controller therefore treats workload features as endogenous and uses smoothing and stability guards to avoid oscillations. A simple stability condition is to limit the rate of change of multipliers: [39]

$$\lambda^{t1} = 1 - \beta \lambda^t \quad \beta \lambda_{\text{raw}}^{t1}, \quad (6.2)$$

with small β , which damps oscillations.

Finally, consider approximation algorithms for selecting compactions under strict constraints. If one were willing to accept randomized schedules, one could apply a dependent rounding scheme to the continuous relaxation to obtain integer decisions with bounded constraint violation in expectation. However, operational systems often prefer deterministic behavior [?]. The presented approach is deterministic given the same inputs, aiding reproducibility, while still allowing controlled exploration when safety margin is large.

7. Distributed Execution Mechanics and Multi-Tenant Considerations

Analytics stores are frequently distributed across many nodes, sharding by key ranges or hash partitions. Compaction is primarily local to each shard because files do not span nodes, but decisions are not independent. Foreground query latency depends on the slowest shard, and shared storage backends or networked block devices can couple interference across nodes [40]. Additionally, multi-tenant deployments require fairness: one tenant’s compaction should not starve others or violate shared budgets.

The policy is implemented as a per-shard controller with optional cluster-level coordination through budget signals. Each shard maintains its own state vector and workload embedding. It enforces local read amplification constraints that are derived from a global target [41]. If a query fans out to N shards and latency is dominated by the maximum, a conservative approach is to allocate each shard a stricter constraint so that the maximum remains bounded. Alternatively, one can allocate constraints dynamically based on shard load and criticality. Let τ_b^{global} be a global per-query bytes-read amplification target aggregated across shards. If queries contact a set $\mathcal{S}_q$ of shards, then an additive model suggests $\sum_{s \in \mathcal{S}_q} \text{RA}_{b,s,q} \leq \tau_b^{\text{global}}$. A practical allocation is to set per-shard targets $\tau_{b,s}$ such that $\sum_s \tau_{b,s} = \tau_b^{\text{global}}$ for typical fanout, then allow rebalancing using shadow prices. If shard s has high compaction cost due to device contention, its λ_w is effectively larger, so it may request a looser target by signaling a higher cost to the coordinator [42]. The coordinator can then adjust $\tau_{b,s}$ within bounds, producing a cluster-level Pareto trade-off between read amplification and compaction cost.

When storage is shared, interference costs are coupled. Let B^{shared} be the shared bandwidth and let shards submit compaction I/O that competes. A distributed primal-dual method can coordinate by broadcasting a shared multiplier $\lambda_{\text{io}}^{\text{shared}}$ that reflects congestion. Each shard includes this multiplier in scoring, reducing compaction aggressiveness when congestion is high. The multiplier can be updated from observed device queueing delay or from a token-bucket rate limiter at the device layer. This approach avoids centralized scheduling of jobs while still enforcing a global budget [43].

Distributed compaction also must respect failure and recovery. Compaction jobs are long-running and may be interrupted. The engine must ensure that partial outputs are either safely discarded or can be resumed. The controller treats partial jobs as sunk cost and updates its state accordingly [?]. To avoid thrashing, it includes hysteresis: if a job was started and partially completed, it is given priority to finish, because abandoning it yields no read benefit but consumes resources. This can be modeled as a continuation bonus in scoring.

In multi-tenant analytics stores, tenants may have different read amplification constraints depending on SLOs. The controller can maintain per-tenant workload embeddings and estimate per-tenant contributions to amplification by tracking which queries belong to which tenant [44]. However, compaction operates on shared files that may contain data from multiple tenants if co-located; in that case, benefits are shared and attribution is complex. A pragmatic approach is to allocate compaction budgets per shard and enforce the strictest relevant read amplification constraint among active tenants, while using weighted objectives to reflect tenant priorities. Let tenant weights be π_k and define a weighted risk-adjusted amplification [45]. Shadow prices then reflect tenant mix and can shift as mix changes.

Network and CPU effects matter for analytics queries. Many queries do not read full values but scan columnar blocks and compute aggregates. Compaction that rewrites files can change compression and encoding choices [46]. For example, rewriting sorted data can improve run-length encoding or dictionary compression, reducing bytes read and CPU per byte. The controller can incorporate a compression benefit term Δ_j^{comp} estimated from observed compression ratios of past compactations on similar data. This connects compaction policy to entropy and coding intuitions: more ordered and clustered data yields lower entropy per symbol for certain encoders, reducing I/O and sometimes CPU. A simple model treats expected compressed size as $S_c = S \cdot \chi H$ where H is an estimated entropy rate and χ is encoder-dependent. Compaction that improves locality can reduce H for secondary columns if correlated with key order, yielding a measurable benefit [47]. Because direct entropy estimation is expensive, the controller uses proxies such as dictionary size and run-length statistics computed during compaction.

8. Evaluation Methodology and Reproducibility Considerations

Evaluating a workload-adaptive compaction policy requires careful methodology because compaction interacts with workload timing, caching, and background variability. A reproducible evaluation must control sources of nondeterminism, report both mean and tail behavior, and include sensitivity analysis to workload shifts and parameter choices.

The primary metrics include bytes-read amplification RA_b , file-probe amplification RA_f , write amplification measured as device bytes written per logical byte ingested, compaction interference measured by foreground latency degradation under controlled load, and resource consumption including CPU and energy proxies. Because the policy enforces a read amplification constraint, evaluation should emphasize constraint satisfaction: the fraction of time windows where $\widehat{RA}_b \varepsilon_t \leq \tau_b$ holds, and the magnitude of any violations. Tail metrics are crucial: report high quantiles of query latency and amplification [?]. In addition, measure adaptation speed after workload shifts by introducing controlled regime changes and measuring the time to restore constraint satisfaction.

Workloads should include point-heavy, scan-heavy, and mixed patterns, as well as realistic analytics traces with time-bounded scans and selective predicates. Because the controller uses workload embeddings, it is important to include non-stationary traces. A standard approach is to run phases: a steady state phase to reach stable level sizes, then introduce a shift such as increasing scan length or changing predicate columns, then observe controller response [?]. The evaluation must ensure that the store is not artificially warmed in a way that hides read amplification differences. Cache should be either controlled (fixed size, warmed consistently) or measured (hit rates reported) to interpret bytes read.

A critical aspect is the baseline [48]. Compare against static leveled compaction with standard size thresholds, tiered compaction with typical parameters, and any hybrid policy available in the engine. Ensure that all baselines are tuned reasonably for the given hardware and workload, but avoid tuning

that uses knowledge unavailable to the adaptive policy. For fairness, allow the adaptive policy to start with conservative parameters and learn online.

Instrumentation error should be quantified [49]. Report the sampling rate m used to estimate $\widehat{RA}_b$, the confidence level $1 - \delta$, and the resulting ε_t . Validate the estimator by periodically computing a more accurate measurement on a subset of windows and comparing. When sketches are used for selectivity and heat, report sketch sizes and error empirically. This is important because the controller’s safety behavior depends on these bounds; overly optimistic error estimates can cause constraint violation, while overly pessimistic estimates can cause unnecessary compaction.

Reproducibility requires determinism in the compaction scheduler as much as possible [50]. Fix random seeds for any exploration. Ensure that compaction thread scheduling is stable or report variability. Use fixed hardware or container resource limits. Record device characteristics such as bandwidth and latency [51]. Because compaction is I/O heavy, small differences in device state can have large effects. If evaluation uses shared devices, isolate experiments or report co-tenant load.

An evaluation should also test robustness to pathological cases. One case is write bursts that create many $L0$ files, increasing read amplification quickly [52]. The controller should respond by prioritizing $L0$ reduction even if it increases write amplification temporarily. Another case is skewed hotspots where a small key interval dominates queries. The controller should focus compaction on that interval, possibly by performing targeted compactations that reduce overlap in hot bins without compacting cold ranges unnecessarily [?]. Another case is high tombstone density due to deletes or updates; the controller should exploit reclaim ratio to reduce write amplification and improve reads.

To interpret trade-offs, present Pareto curves between read amplification and write amplification under different constraint thresholds τ_b and different endurance budgets. Even without explicit plots, the evaluation should report how tightening τ_b increases write cost and whether the increase is smooth or exhibits thresholds due to level structure. A well-behaved controller should degrade gracefully: as constraints tighten, it should allocate more compaction to hot ranges and higher-overlap levels first, rather than uniformly increasing compaction everywhere [53].

Finally, evaluation should examine query plan interactions. If the analytics engine changes scan parallelism or shard fanout in response to latency, the workload seen by each shard changes. To isolate compaction effects, either fix query plans or report plan changes. If plan adaptation is enabled, treat it as part of the system and report combined behavior, but then the evaluation should include cases where plan feedback could cause oscillation [54]. The controller’s smoothing and multiplier damping should prevent runaway oscillations, which can be validated by measuring frequency content of λ_{ra} and $\widehat{RA}_b$ over time and ensuring that they do not exhibit persistent cycles.

9. Conclusion

This paper described a workload-adaptive compaction policy for LSM-based analytics stores with explicit read amplification constraints. The design treats compaction as an online constrained control problem, integrating workload embeddings, storage-state signals, and a marginal benefit model for candidate compaction jobs. A primal-dual scoring mechanism with safety margins derived from estimator error enforces read amplification constraints conservatively, while multi-objective costs capture write amplification, interference, and resource budgets. The policy respects LSM storage internals by operating through candidate generation, interval-structured conflict management, and efficient data structures such as interval trees and low-rank workload projections [55]. The approach incorporates approximation-aware instrumentation, uncertainty penalties, and stabilization mechanisms to handle delayed effects and non-stationary workloads. Distributed execution is addressed through per-shard controllers coordinated by shared budget multipliers, enabling scalability and multi-tenant fairness without centralized job scheduling. Evaluation considerations emphasize constraint satisfaction, tail behavior, robustness to workload shifts, and reproducibility under realistic system nondeterminism. The overall result is a

compaction control framework that can adapt compaction effort to workload dynamics while maintaining explicit bounds on read amplification, providing a principled basis for tuning and extension in analytics-focused LSM deployments [?].

References

- [1] M. Zaccarini, F. Poltronieri, D. Borsatti, W. Cerroni, L. Foschini, G. Y. Grabarnik, D. Scotece, L. Shwartz, C. Stefanelli, and M. Tortonesi, “Chaos engineering based kubernetes pod rescheduling through deep sets and reinforcement learning,” in *NOMS 2025-2025 IEEE Network Operations and Management Symposium*, pp. 1–7, IEEE, 5 2025.
- [2] F. Mora, A. Desai, E. Polgreen, and S. A. Seshia, “Message chains for distributed system verification,” *Proceedings of the ACM on Programming Languages*, vol. 7, pp. 2224–2250, 10 2023.
- [3] J. Yang, Z. Wang, R. Chen, and H. Chen, “A system-level abstraction and service for flourishing ai-powered applications,” in *Proceedings of the 16th ACM SIGOPS Asia-Pacific Workshop on Systems*, pp. 106–114, ACM, 10 2025.
- [4] R. Malik, S. Kim, X. Jin, C. Ramachandran, J. Han, I. Gupta, and K. Nahrstedt, “Mlr-index: An index structure for fast and scalable similarity search in high dimensions,” in *International Conference on Scientific and Statistical Database Management*, pp. 167–184, Springer, 2009.
- [5] Y. Qian, S. Ying, and B. Wang, “Smc - anomaly detection in distributed systems via variational autoencoders,” in *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 2822–2829, IEEE, 10 2020.
- [6] *Blockchain for Distributed Systems Security*. Wiley, 4 2019.
- [7] B. Savenko, A. Kashtalian, S. Lysenko, and O. Savenko, “Malware detection by distributed systems with partial centralization,” in *2023 IEEE 12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, pp. 265–270, IEEE, 9 2023.
- [8] “Uml language in distributed system modeling,” *Distributed Processing System*, vol. 1, 3 2020.
- [9] A. B. Chernyshev, V. Antonov, and Z. Mayransaev, “Unified sectoral criterion for the stability of distributed systems,” in *Proceedings of X All-Russian Scientific Conference “System Synthesis and Applied Synergetics”*, pp. 351–355, Southern Federal University, 10 2021.
- [10] Y. Tyryshkina, “Understanding join strategies in distributed systems,” in *2021 International Seminar on Electron Devices Design and Production (SED)*, pp. 1–4, IEEE, 4 2021.
- [11] Y. Dong and Z. Mi, “Ioguard: Software-based i/o page fault handling with one cpu core,” in *Proceedings of the 15th Asia-Pacific Symposium on Internetwork*, pp. 337–346, ACM, 7 2024.
- [12] T. Srinivasan, R. Chandrasekar, V. Vijaykumar, V. Mahadevan, A. Meyyappan, and A. Manikandan, “Localized tree change multicast protocol for mobile ad hoc networks,” in *2006 International Conference on Wireless and Mobile Communications (ICWMC’06)*, pp. 44–44, IEEE, 2006.
- [13] T. Peroutka, I. Murturi, P. K. Donta, and S. Dustdar, “A graph-based approach to human activity recognition,” in *2024 IEEE 6th International Conference on Cognitive Machine Intelligence (CogMI)*, pp. 117–126, IEEE, 10 2024.
- [14] M. Anisetti, C. A. Ardagna, E. Damiani, and N. E. Ioini, *Certification of Modern Distributed Systems*, pp. 41–60. Springer International Publishing, 7 2024.
- [15] . Ευθυμιάδης and . Χαλδαίος, “Ανάπτυξη κατανεμημένου συστήματος σε υπολογιστικό νέφος,” 6 2018.
- [16] W. N. S. W. Nik, “A survey on data replication and resource management in distributed systems,” *International Journal of Advanced Trends in Computer Science and Engineering*, pp. 2638–2646, 10 2019.
- [17] S. He, C. Fu, G. Feng, J. Lv, and F. Deng, “Singular value manipulating: An effective drl-based adversarial attack on deep convolutional neural network,” *Neural Processing Letters*, vol. 55, pp. 12459–12480, 10 2023.
- [18] R. Chandrasekar, R. Suresh, and S. Ponnambalam, “Evaluating an obstacle avoidance strategy to ant colony optimization algorithm for classification in event logs,” in *2006 International Conference on Advanced Computing and Communications*, pp. 628–629, IEEE, 2006.

- [19] J. A. McDougall, A. Brighente, W. Großmann, B. A. McDougall, J. Stock, and H. Federrath, “Love is in the air - location verification of ads-b signals using distributed public sensors,” in *ICC 2023 - IEEE International Conference on Communications*, pp. 6040–6045, IEEE, 5 2023.
- [20] M. Witter and A. R. D. Vit, “Blockchain and distributed systems: Basic concepts and implications,” 3 2024.
- [21] P. Dubey, D. Srivastava, K. Singh, and M. V. Singh, “Middleware architecture for microservices based distributed system,” in *2023 13th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, pp. 200–207, IEEE, 1 2023.
- [22] X. Song, R. Chen, H. Song, Y. Zhang, and H. Chen, “Unified and near-optimal multi-gpu cache for embedding-based deep learning,” *ACM Transactions on Computer Systems*, vol. 44, pp. 1–32, 11 2025.
- [23] E. Arif, “Implementing a distributed system as a solution to problems in the indosat cooperative,” *JURNAL PETIK*, vol. 5, pp. 24–29, 10 2019.
- [24] Y. Li, Z. Zhou, B. Sedlak, and S. Dustdar, “Mgg-ad: Multi-granularity graph-based anomaly detection in iot systems,” in *2025 IEEE International Conference on Web Services (ICWS)*, pp. 1–10, IEEE, 7 2025.
- [25] “Three-layer distributed system based on bayesian classifier,” *Distributed Processing System*, vol. 3, 10 2022.
- [26] F. Kelbert and A. Pretschner, “Data usage control for distributed systems,” *ACM Transactions on Privacy and Security*, vol. 21, pp. 12–32, 4 2018.
- [27] J. Stein, T. Rohe, F. Nappi, J. Hager, D. Bucher, M. Zorn, M. Kölle, and C. Linnhoff-Popien, “Introducing reduced-width qnns, an ai-inspired ansatz design pattern,” 1 2023.
- [28] N. Desai, “Modeling reliability for distributed systems,” 1 2020.
- [29] I. Kunze, D. Scheurenberg, L. Tirpitz, S. Geisler, and K. Wehrle, “In-situ model validation for continuous processes using in-network computing,” in *2024 IEEE 7th International Conference on Industrial Cyber-Physical Systems (ICPS)*, pp. 1–6, IEEE, 5 2024.
- [30] R. Chandrasekar and T. Srinivasan, “An improved probabilistic ant based clustering for distributed databases,” in *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI*, pp. 2701–2706, 2007.
- [31] *OSDI - Retrofitting High Availability Mechanism to Tame Hybrid Transaction/Analytical Processing*, 7 2021.
- [32] M. Jancic, F. Strnisa, and G. Kosec, “Implicit-explicit error indicator based on approximation order,” in *2022 7th International Conference on Smart and Sustainable Technologies (SpliTech)*, pp. 1–4, IEEE, 7 2022.
- [33] L. A. Vlasenko, A. G. Rutkas, and A. A. Chikrii, “Stochastic differential games in distributed systems with delay,” *Journal of Automation and Information sciences*, vol. 1, pp. 41–54, 1 2021.
- [34] M. Wu, L. Mao, Y. Lin, Y. Jin, Z. Li, H. Lyu, J. Tang, X. Lu, H. Tang, D. Dong, H. Chen, and B. Zang, “Jade: A high-throughput concurrent copying garbage collector,” in *Proceedings of the Nineteenth European Conference on Computer Systems*, pp. 1160–1174, ACM, 4 2024.
- [35] A. K. Chopra, S. H. C. V, and M. P. Singh, *Multiagent Foundations for Distributed Systems: A Vision*, pp. 62–71. Germany: Springer International Publishing, 3 2022.
- [36] L. Colombi, S. Dahdal, E. D. Caro, R. Fronteddu, A. Gilli, A. Morelli, F. Poltronieri, M. Tortonesi, N. Suri, and C. Stefanelli, “Efficient data dissemination via semantic filtering at the tactical edge,” in *MILCOM 2024 - 2024 IEEE Military Communications Conference (MILCOM)*, pp. 457–462, IEEE, 10 2024.
- [37] M. Huisman and C. Seculeanu, *ISO LA (1) - Verification and Validation of Concurrent and Distributed Systems (Track Summary)*, pp. 421–425. Germany: Springer International Publishing, 10 2020.
- [38] I. Čilić, A. Lackinger, P. A. Frangoudis, I. P. Žarko, A. Furutanpey, I. Murturi, and S. Dustdar, “Reactive orchestration for hierarchical federated learning under a communication cost budget,” in *2025 IEEE International Conference on Machine Learning for Communication and Networking (ICMLCN)*, pp. 1–7, IEEE, 5 2025.
- [39] K. W. HamaAli and S. R. M. Zeebaree, “Resources allocation for distributed systems: A review,” 1 2021.
- [40] V. Vijaykumar, R. Chandrasekar, and T. Srinivasan, “An obstacle avoidance strategy to ant colony optimization algorithm for classification in event logs,” in *2006 IEEE Conference on Cybernetics and Intelligent Systems*, pp. 1–6, IEEE, 2006.

- [41] O. Scekcic, M. Riveni, H.-L. Truong, and S. Dustdar, *Social Interaction Analysis for Team Collaboration*, pp. 2607–2621. Springer New York, 6 2018.
- [42] J. W. Baker, B. Bradley, and P. J. Stafford, *Spatially Distributed Systems*, pp. 447–482. Cambridge University Press, 10 2021.
- [43] D. Lindsay, S. S. Gill, D. Smirnova, and P. Garraghan, “The evolution of distributed computing systems: from fundamental to new frontiers,” *Computing*, vol. 103, pp. 1859–1878, 1 2021.
- [44] A. Girault, G. Gössler, R. Guerraoui, J. Hamza, and D.-A. Seredinschi, *FORTE - Monotonic Prefix Consistency in Distributed Systems*, pp. 41–57. Germany: Springer International Publishing, 5 2018.
- [45] M. Zaccarini, F. Poltronieri, C. Stefanelli, and M. Tortonesi, “Hybridized hot restart via reinforcement learning for microservice orchestration,” in *NOMS 2025-2025 IEEE Network Operations and Management Symposium*, pp. 1–7, IEEE, 5 2025.
- [46] R. Walther, C. Weinhold, P. Anthor, and M. Roitzsch, “Multi-stakeholder policy enforcement for distributed systems,” in *Proceedings of the 10th International Workshop on Container Technologies and Container Clouds*, pp. 7–12, ACM, 12 2024.
- [47] A. Biswas and A. K. Tripathi, *Preselection Based Leader Election in Distributed Systems*, pp. 261–271. Germany: Springer International Publishing, 5 2022.
- [48] I. Schagaev and S. Farrell, *Distributed Systems: Resilience, Desperation*, pp. 343–368. Springer International Publishing, 7 2024.
- [49] I. Schagaev, *Distributed Systems: Maximizing Resilience*, pp. 249–266. Springer International Publishing, 7 2019.
- [50] A. Samanta, D. Karnehm, A. Neve, and S. Williamson, “Data-informed healthy cell clustering technique for second-life applications of retired electric vehicle batteries,” in *2024 IEEE Transportation Electrification Conference and Expo (ITEC)*, pp. 1–5, IEEE, 6 2024.
- [51] A. Hojjat, J. Haberer, T. Zainab, and O. Landsiedel, “Limitnet: Progressive, content-aware image offloading for extremely weak devices & networks,” in *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services*, pp. 519–533, ACM, 6 2024.
- [52] D. G. Balreira, T. da Silva Araújo, and F. Petrillo, “Visualizing kubernetes distributed systems: An exploratory study,” in *2023 IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 12–22, IEEE, 10 2023.
- [53] R. Malik, C. Ramachandran, I. Gupta, and K. Nahrstedt, “Samera: a scalable and memory-efficient feature extraction algorithm for short 3d video segments,” in *IMMERSCOM*, p. 18, 2009.
- [54] A. A. Hussein, A. AL-Zebari, N. Omar, K. J. Merceedi, A. J. Ahmed, N. O. M. Salim, S. S. Hasan, S. F. Kak, I. M. Ibrahim, H. M. Yasin, and A. A. Salih, “State of art survey for fault tolerance feasibility in distributed systems,” *Asian Journal of Research in Computer Science*, pp. 19–34, 9 2021.
- [55] S. Majumdar, *Resource Management on Distributed Systems*. Wiley, 9 2024.