Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty

Andy Huynh
Boston University
ndhuynh@bu.edu

Harshal A. Chaudhari
Boston University
harshal@bu.edu

Evimaria Terzi
Boston University
evimaria@bu.edu

Manos Athanassoulis
Boston University
mathan@bu.edu

ABSTRACT

Log-Structured Merge trees (LSM trees) are increasingly used as the storage engines behind several data systems, frequently deployed in the cloud. Similar to other database architectures, LSM trees consider information about the *expected* workload (e.g., reads vs. writes, point vs. range queries) to optimize their performance via tuning. However, operating in a shared infrastructure like the cloud comes with workload *uncertainty* due to the fast-evolving nature of modern applications. Systems with static tuning discount the variability of such hybrid workloads and hence provide an inconsistent and overall suboptimal performance.

To address this problem, we introduce Endure – a new paradigm for tuning LSM trees in the presence of workload uncertainty. Specifically, we focus on the impact of the choice of compaction policies, size ratio, and memory allocation on the overall performance. Endure considers a robust formulation of the throughput maximization problem and recommends a tuning that maximizes the worst-case throughput over the *neighborhood* of each expected workload. Additionally, an uncertainty tuning parameter controls the size of this neighborhood, thereby allowing the output tunings to be conservative or optimistic. Through both model-based and extensive experimental evaluations of Endure in the state-of-the-art LSM-based storage engine, RocksDB, we show that the robust tuning methodology consistently outperforms classical tuning strategies. The robust tunings output by Endure lead up to a 5× improvement in throughput in the presence of uncertainty. On the flip side, Endure tunings have negligible performance loss when the observed workload exactly matches the expected one.

PVLDB Reference Format:
Andy Huynh, Harshal A. Chaudhari, Evimaria Terzi, and Manos Athanassoulis. Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty. PVLDB, 15(8): 1605-1618, 2022.
doi:10.14778/3529337.3529345

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/BU-DiSC/endure.

1 INTRODUCTION

Ubiquitous LSM-based Key-Value Stores. Log-Structured Merge trees (LSM trees) are the most commonly deployed data structures used in the backend storage of modern key-value stores [59]. LSM trees offer a high ingestion rate and fast reads, making them widely adopted by systems such as RocksDB [31] at Facebook, LevelDB [33] and BigTable [18] at Google, HBase [37], Cassandra [8] at Apache, WiredTiger [80] at MongoDB, X-Engine [40] at Alibaba, and DynamoDB [29] at Amazon.

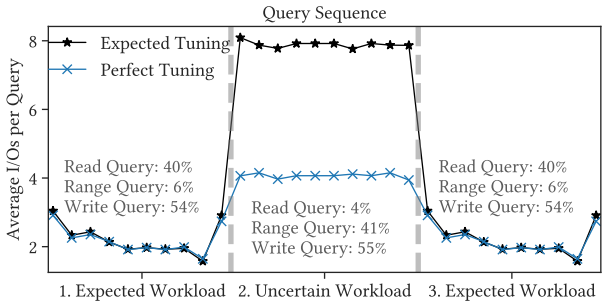


Figure 1: LSM tree tunings and their performance on observed workloads. While both workloads have a similar ratio of reads and writes, the uncertain workload has a higher percentage of range queries causing the expected system tuning to experience a 2× degradation in performance.

LSM trees store incoming data in a memory buffer, which is flushed to storage when it is full and merged with earlier flushed buffers to form a collection of sorted runs with exponentially increasing sizes [53]. Frequent merging of sorted runs leads to higher merging costs but facilitates faster lookups (*leveling*). On the flip side, lazy merging policies (*tiering*) trade lookup performance for lower merging costs [67].

Tuning LSM trees. As the number of applications relying on LSM-based storage backends increases, the problem of performance tuning for LSM trees has garnered a lot of attention. A common assumption made by all these methods is that one has *complete knowledge about the expected workload and the execution environment.* Given such knowledge, prior work optimizes memory allocation to Bloom filters across different levels, memory distribution between the buffers and the Bloom filters, and the choice of merging policies (i.e., *leveling* or *tiering*) [26]. Different optimization objectives have led to hybrid merging policies with more fine-grained tunings [27, 28, 42], optimized memory allocation strategies [14, 49, 51], variations of Bloom filters [55, 83], new compaction routines [5, 52, 66, 67, 84], and exploitation of data characteristics [1, 64, 82].

The Only Certainty is Uncertainty. Even when accurate information about the workload and underlying hardware is available, tuning data systems is a notoriously difficult research problem [19, 22, 73]. The explosive growth in the use of the cloud infrastructure for data management [36, 48, 65] has exacerbated this

Proceedings of the VLDB Endowment, Vol. 15, No. 8 ISSN 2150-8097.
doi:10.14778/3529337.3529345

problem due to increased uncertainty and variability in workloads [23, 32, 38, 39, 57, 60, 63, 68–70, 81].

An Example. Before describing our framework, we give an example that depicts how variation in observed workloads relative to the expected workload – used during tuning of the LSM tree-based storage – leads to suboptimal performance. In Figure 1, the x-axis shows a sequence of workloads executed over an LSM-based engine, while the y-axis shows the average disk accesses per workload. The experiment is split into three sessions – the first and the last sessions receive the expected workload, while the second session receives a different workload. Although it has the same reads vs. writes ratio as the expected workload, it has a higher percentage of short range queries in comparison to the point queries. The solid black line shows the performance of a system tuned for the expected workload. Note that the average I/Os increase dramatically in the second session even though the amount of data being read is approximately the same. On the other hand, the blue line corresponds to each session having its ideal tuning, leading to only half as many I/Os per operation. Note that it is not feasible to continually change tunings during execution, as it requires redistribution of the allocated memory between different components of the tree and potentially changing its shape. Hence, we want to *find a tuning that is close to optimal for both the expected and the observed workload.*

Our Work: Robust LSM Tree Tuning. To address this suboptimality caused by variations in the observed workload, we depart from the classical view of database tunings which assumes accurate knowledge about the expected workload. Rather, we introduce Endure, a new *robust tuning paradigm* that incorporates expected uncertainty into optimization and apply it to LSM trees.

We formulate the Robust Tuning problem that seeks an LSM tree configuration that maximizes the worst-case throughput over all the workloads in the *neighborhood* of an expected workload. We use the notion of KL-divergence between probability distributions to define the neighborhood size, implicitly assuming that the uncertain workloads would be contained in the neighborhood. As the KL-divergence boundary condition approaches zero, our problem becomes equivalent to the classical optimization problem (henceforth referred to as the Nominal Tuning problem). More specifically, our approach uses as input the expected size of the uncertainty neighborhood, which dictates the qualitative characteristics of the solution. Intuitively, the larger the size of the uncertainty neighborhood, the larger the workload discrepancy a robust tuning can accommodate. Leveraging work on robust optimization from the Operations Research community [10–12], we efficiently solve the Robust Tuning problem and find the robust tuning for LSM tree-based storage systems. A similar problem of using workload uncertainty while determining the physical design of column-stores has been explored in prior work [58]. However, their methodology is not well suited for the LSM tuning problem. We provide additional details regarding this in Section 9.

Contributions. To the best of our knowledge, our work presents the first systematic approach for robust tuning of LSM tree-based key-value stores under workload uncertainty. Our technical and practical contributions can be summarized as follows:

- We incorporate workload uncertainty in LSM tuning and show how to find a robust tuning efficiently. Our algorithm can be tuned for varying degrees of workload uncertainty, and is simple enough to be adopted by current state-of-the-art LSM storage engines (§4 and §5).
- We developed an uncertainty benchmark that can test the robustness of current state-of-the-art database systems (§6).
- In our model-based analysis, we show that robust tunings from Endure provide up to 5× higher throughput when faced with uncertain workloads (§7).
- We integrate and test Endure on RocksDB, a state-of-the-art LSM storage engine, to show the feasibility of robust tuning on commercial systems. We show Endure achieves up to 2.4× throughput speedups, our algorithm works regardless of database size (§8).
- To encourage reproducible research, we make our robust tuning framework publicly available [7].

2 BACKGROUND ON LSM TREES

Basics. LSM trees use the *out-of-place* ingestion paradigm to store key-value pairs. Writes, updates or deletes are placed in a memory buffer. Once full, its contents are sorted based on the key, forming an *immutable sorted run*, then flushed to secondary storage. Sorted runs are organized in levels. Thus for an LSM tree with L disk-resident levels, we denote the memory buffer as Level 0, and label the remaining levels in storage from 1 to L. The disk-resident sorted runs have exponentially increasing sizes via a tunable size ratio T.

We denote the number of bits of main memory allocated to the buffer as m_{buf}, which holds a number of entries with a fixed entry size E. For example, in RocksDB the default buffer size is m_{buf} = 64MB, and depending on the application, the entry size typically varies between 64B and 1KB. This buffer at Level 0 can be updated in-place, however, runs in levels 1 and beyond are immutable. Each level i has a capacity threshold of $(T-1)T^{i-1} \cdot \frac{m_{\text{buf}}}{E}$ entries, thus, level capacities are exponentially increasing by a factor of T. The total number of levels L for a given T is

$$L(T) = \left\lceil \log_T \left(\frac{N \cdot E}{m_{\text{buf}}} + 1 \right) \right\rceil, \tag{1}$$

where N is the total number of entries across all levels [26, 55, 66].

Compaction Policies: Leveling and Tiering. Classically, LSM trees support two merging policies: leveling and tiering. In leveling, each level may have at most one run, and every time a run in Level $i-1$ ($i \geq 1$) is moved to Level i, it is greedily sort-merged (compaction) with the run from Level i, if it exists. With tiering, every level must accumulate T runs before they trigger a compaction. During a compaction, entries with a matching key are consolidated and only the most recent valid entry is retained [30, 59]. Recently hybrid compaction policies fuse leveling and tiering in a single tree to strike a balance between the read and write throughput [27, 28].

LSM tree Operations. An LSM tree supports: (a) writes of new key-value pairs, (b) point queries, and (c) range queries.

Writes: All write operations are handled by a buffer append. Once the buffer is full, a compaction is triggered. Any write may include either a new key-value pair, an existing key that *updates* its value, or a special entry that *deletes* an existing key.

Point Queries: A point query searches for the value of a specific unique key. It begins by looking at the memory buffer, then traverses the tree from the smallest to the largest level. For tiering, at each level a lookup moves from the most to the least recent tier. The lookup then terminates when it finds the first matching entry. Note that a point query might return either an *empty* or a *non-empty* result. We differentiate the two because workloads with empty point queries can be further optimized [26].

Range Queries: A range lookup returns the most recent versions of the target keys by sort-merging all qualifying runs from the tree.

Optimizing Lookups. Read performance is optimized using Bloom filters and fence pointers. In the worst-case, a lookup needs to probe every run. To reduce this cost, LSM engines use one Bloom filter per run in main memory [26, 31]. Bloom filters [13] are probabilistic membership test data structures that exhibit a false positive f as a function of the ratio between the memory allocated m_{filt} to them and the elements it indexes. In LSM trees, Bloom filters allow a lookup to skip probing a run altogether if the filter-lookup returns negative. In practice, for efficient storage, Bloom filters are maintained at the granularity of files [30]. Fence pointers store the smallest key per disk page in memory [26], to quickly identify which page(s) to read for a lookup, and perform up to one I/O per run for point lookups.

Tuning LSM Trees. Prior to this work, efforts to systematically tune LSM trees assume that the workload information and the execution environment are accurately known. Under that assumption, the main focus on LSM tuning has been on deciding how to allocate the available main memory between Bloom filters and buffering [49, 51], while often the size ratio and the merging strategy was also co-tuned [26] Such design decisions are common across industry standard LSM-based engines such as Apache Cassandra [8], AsterixDB [6], RocksDB [31], and InfluxDB [47]. In addition, recent work has introduced new hybrid merging strategies [27, 28, 67], and optimizations for faster data ingestion [54] and performance stability [52].

3 PROBLEM DEFINITIONS

In this section, we provide the formal problem definitions on how to choose the *design parameters* of an LSM tree. Before proceeding, we give a brief introduction to our notation.

3.1 Notation

As we discussed above, LSM trees have two types of parameters: the *design parameters* that are changed primarily for performance, and the *system parameters* that are given and therefore untunable.

Design Parameters. The design parameters we consider in this paper are the size ratio (T), the memory allocated to the Bloom filters (m_{filt}), the memory allocated to the write buffer (m_{buf}) and the compaction policy (π). These are ubiquitous design parameters and have been extensively studied as having the largest impact on performance [26, 53]. Therefore, we focus on these parameters in order to define a problem that is agnostic to the LSM engine used. Recall that the policy refers to either leveling or tiering.

System Parameters. A complex data structure like an LSM tree also has various *system parameters* and other non-tunable ones (e.g., total memory (m), data entry size E, page size B, data size N).

Table 1: Summary of problem notation

Type	Term	Definition
Design	m_{filt}	Memory allocated for Bloom filters
	m_{buf}	Memory allocated for the write buffer
	T	Size ratio between consecutive levels
	π	Compaction policy (*tiering*/*leveling*)
System	m	Total memory (filters+buffer) ($m = m_{\text{buf}} + m_{\text{filt}}$)
	E	Size of a key-value entry
	B	Number of entries that fit in a page
	N	Total number of entries
Workload	z_0	Percentage of zero-result point lookups
	z_1	Percentage of non-zero-result point lookups
	q	Percentage of range queries
	w	Percentage of writes

LSM Tree Configuration. We use Φ to denote the LSM tree tuning configuration which describes the values of the tunable parameters together $\Phi := (T, m_{\text{filt}}, \pi)$. Note that we only use the memory for Bloom filters m_{filt} and not m_{buf}, because the latter can be derived using the former and total available memory: $m_{\text{buf}} = m - m_{\text{filt}}$.

Workload. The choice of the parameters in Φ depends on the input (expected) workload, i.e., the fraction of empty lookups (z_0), non-empty lookups (z_1), range lookups (q), and write (w) queries within an observation period. Such a period can be either defined either over a fixed time interval, or over a certain number of queries. Note that this workload representation is common for analyzing and tuning LSM trees [26, 53]. Additionally, complex workloads (i.e., SQL statements) generate access patterns on the storage engine and can be broken down into the same basic operations. This mapping of complex queries to basic operations is also common for performance tuning of LSM tree-based storage engines [17]. Therefore a workload can be expressed as a vector $\mathbf{w} = (z_0, z_1, q, w)^\intercal \geq 0$ describing the proportions of the different kinds of queries. Clearly, $z_0 + z_1 + q + w = 1$ or alternatively: $\mathbf{w}^\intercal \mathbf{e} = 1$ where $\mathbf{e}$ denotes a column vector of ones.

Each type of query (non-empty lookups, empty lookups, range lookups and writes) has a different cost, denoted as $Z_0(\Phi)$, $Z_1(\Phi)$, $Q(\Phi)$, $W(\Phi)$, as there is a dependency between the cost of each type of query and the design Φ. For ease of notation, we use $\mathbf{c}(\Phi) = (Z_0(\Phi), Z_1(\Phi), Q(\Phi), W(\Phi))^\intercal$ to denote the vector of the costs of executing different types of queries. Thus, given a specific configuration (Φ) and a workload ($\mathbf{w}$), the expected cost for the workload can be computed as:

$$\mathrm{C}(\mathbf{w}, \Phi) = \mathbf{w}^\intercal \mathbf{c}(\Phi) = z_1 \cdot Z_0(\Phi) + z_0 \cdot Z_1(\Phi) + q \cdot Q(\Phi) + w \cdot W(\Phi). \quad (2)$$

We present a summary of all of our notation in Table 1.

3.2 The Nominal Tuning Problem

Traditionally, the designers have focused on finding the configuration Φ^* that minimizes the total cost $\mathrm{C}(\mathbf{w}, \Phi^*)$, for a given fixed workload $\mathbf{w}$. We call this problem the Nominal Tuning problem, defined as follows:

Problem 1 (Nominal Tuning). *Given fixed* $\mathbf{w}$ *find the tuning configuration of the LSM tree* Φ_N *such that*

$$\Phi_N = \arg\min_{\Phi} C(\mathbf{w}, \Phi). \tag{3}$$

The nominal tuning problem described above captures the classical tuning paradigm. It uses a cost-model to find a system configuration that minimizes the cost given a specific workload and system environment. Specifically, prior tuning approaches for LSM trees solve the nominal tuning problem when proposing optimal memory allocation, and merging policies [26, 51].

3.3 The Robust Tuning Problem

In this work, we attempt to compute high-performance configurations that minimize the expected cost of operation, as expressed in Equation (2), in the presence of uncertainty with respect to the expected workload.

The Nominal Tuning problem assumes perfect information about the workload for which to tune the system. For example, we may assume that the input vector $\mathbf{w}$ represents the workload for which we optimize, while in practice, $\mathbf{w}$ is simply an estimate of what the workload will look like. Hence, the configuration obtained by solving Problem 1 may result in high variability in the system performance that will inevitably depend on the actual observed workload upon the deployment of the system.

We capture this uncertainty by reformulating Problem 1 to take into account the variability that can be observed in the input workload. Given an expected workload $\mathbf{w}$, we introduce the notion of the *uncertainty region* of $\mathbf{w}$, which we denote by $\mathcal{U}_{\mathbf{w}}$.

We can define the robust version of Problem 1, under the assumption that there is uncertainty in the input workload as follows:

Problem 2 (Robust Tuning). *Given* $\mathbf{w}$ *and uncertainty region* $\mathcal{U}_{\mathbf{w}}$ *find the tuning configuration of the LSM tree* Φ_R *such that*

$$\begin{aligned} \Phi_R \quad &= \quad \arg\min_{\Phi} C(\hat{\mathbf{w}}, \Phi) \\ \textit{s.t.,} \quad &\quad \hat{\mathbf{w}} \in \mathcal{U}_{\mathbf{w}}. \end{aligned} \tag{4}$$

Note that the above problem definition intuitively states the following: it recognizes that the input workload $\mathbf{w}$ will not be observed exactly, and it assumes that any workload in $\mathcal{U}_{\mathbf{w}}$ is possible. Then, it searches for the configuration $\Phi_{\mathbf{w}}$ that is best for the *worst-case* scenario among all those in $\mathcal{U}_{\mathbf{w}}$.

The challenge in solving Robust Tuning is that one needs to explore all the workloads in the uncertainty region in order to solve the problem. In the next section, we show that this is not necessary. In fact, by appropriately rewriting the problem definition we show that we can solve Problem 2 in polynomial time.

4 ALGORITHMS FOR ROBUST TUNING

In this section, we discuss our solutions to the Robust Tuning problem. On a high level, the solution strategy is the following: first, we express the objective of the problem (as expressed in Equation (4)) as a standard continuous optimization problem. We then take the *dual* of this problem and use existing results in robust optimization to show: (i) the duality gap between the primal and the dual is zero, and (ii) the dual problem is solvable in polynomial time. Thus, the dual solution can be translated into the optimal

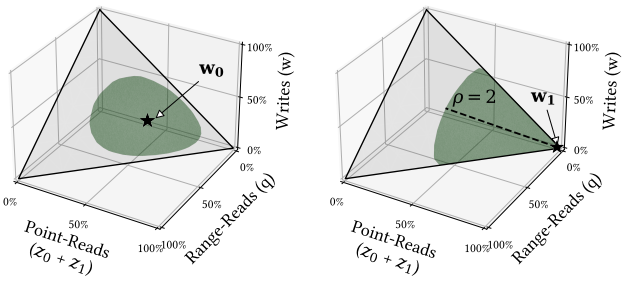


Figure 2: Workload uncertainty neighborhoods ($\mathcal{U}_{\mathbf{w}}$), denoted by the green shaded region, for two different expected workloads ($\mathbf{w}$) and ρ.

solution for the primal, i.e., the original Robust Tuning problem. The specifics of the methodology are described below:

Defining the Uncertainty Region $\mathcal{U}_{\mathbf{w}}$. Recall that $\mathbf{w}$ is a probability vector, i.e., $\mathbf{w}^\intercal \mathbf{e} = 1$. Thus, in order to define the uncertainty region $\mathcal{U}_{\mathbf{w}}$, we use the Kullback-Leibler (KL) divergence function [50] defined as follows:

Definition 1. *The KL-divergence distance between two probability distributions* $\vec{p} = (p_1, \cdots, p_m)^\intercal \geq 0$ *and* $\vec{q} = (q_1, \cdots, q_m)^\intercal \geq 0$ *is defined as,*

$$I_{KL}(\vec{p}, \vec{q}) = \sum_{i=1}^{m} p_i \log\left(\frac{p_i}{q_i}\right).$$

Since our workloads are represented as probability distributions, the KL-divergence is the most natural choice of distance between them. One could use L_p norms instead. However, calculating the L_p norm between workloads requires a summation of the p^{th} power of differences in probabilities, which are extremely small values, which are not meaningful in this setting.

Using the KL-divergence we can now formalize the uncertainty region around an expected workload $\mathbf{w}$ as follows,

$$\mathcal{U}_{\mathbf{w}}^{\rho} = \{\hat{\mathbf{w}} \in \mathbb{R}^4 \mid \hat{\mathbf{w}} \geq 0, \hat{\mathbf{w}}^\intercal \mathbf{e} = 1, I_{KL}(\hat{\mathbf{w}}, \mathbf{w}) \leq \rho\}. \tag{5}$$

Here, ρ determines the maximum KL-divergence that is allowed between any workload $\hat{\mathbf{w}}$ in the uncertainty region and the input expected workload $\mathbf{w}$. Note that the definition of the uncertainty region takes as input the parameter ρ, which intuitively defines the neighborhood around the expected workload. Figure 2 shows an example of the uncertainty region for $\rho = 0.2$ and expected workload $\mathbf{w}_0 = (25\%, 25\%, 25\%, 25\%)$, and for $\rho = 2$ and expected workload $\mathbf{w}_1 = (97\%, 1\%, 1\%, 1\%)$. For this visualization, we combined the two types of read queries (empty and non-empty) onto one axis. Note that the shape of the uncertainty region is defined by the expected workload, the value of ρ, and the fact that all workloads are restricted to be probability distributions. In terms of notation, ρ is required for defining the uncertainty region $\mathcal{U}_{\mathbf{w}}^{\rho}$. However, we drop the superscript notation unless required for context.

Rewriting of the ROBUST TUNING Problem (Primal). Using the above definition of the workload uncertainty region $\mathcal{U}_{\mathbf{w}}^{\rho}$, we are now ready to proceed to the solution of the Robust Tuning problem. For a given ρ, the problem definition, as captured by

Equation (4), can be rewritten as follows:

$$\min_{\Phi} \max_{\hat{\mathbf{w}} \in \mathcal{U}_{\mathbf{w}}^{\rho}} \hat{\mathbf{w}}^{\intercal} \mathbf{c}(\Phi). \tag{6}$$

This rewrite captures the intuition that the optimization is done over the *worst-case* scenario across all the workloads in the uncertainty region $\mathcal{U}_{\mathbf{w}}$. Equation (6) can be rewritten by introducing an additional variable $\beta \in \mathbb{R}$, as follows:

$$\begin{aligned} \min_{\beta,\Phi} \quad & \beta \\ \text{s.t.,} \quad & \hat{\mathbf{w}}^{\intercal} \mathbf{c}(\Phi) \leq \beta \quad \forall \hat{\mathbf{w}} \in \mathcal{U}_{\mathbf{w}}. \end{aligned} \tag{7}$$

This reformulation allows us to remove the min max term in the objective from Equation (6). The constraint in Equation (7) can be equivalently expressed as,

$$\begin{aligned} \beta \quad &\geq \quad \max_{\hat{\mathbf{w}}} \left\{ \hat{\mathbf{w}}^{\intercal} \mathbf{c}(\Phi) | \hat{\mathbf{w}} \in \mathcal{U}_{\mathbf{w}} \right\} \\ &= \quad \max_{\hat{\mathbf{w}} \geq 0} \left\{ \hat{\mathbf{w}}^{\intercal} \mathbf{c}(\Phi) \middle| \hat{\mathbf{w}}^{\intercal} \mathbf{e} = 1, \sum_{i=1}^{m} \hat{w}_i \log\left(\frac{\hat{w}_i}{w_i}\right) \leq \rho \right\}. \end{aligned}$$

Finally, the Lagrange function for the optimization on the right-hand side of the above equation is:

$$\mathcal{L}(\hat{\mathbf{w}}, \lambda, \eta) = \hat{\mathbf{w}}^{\intercal} \mathbf{c}(\Phi) + \rho\lambda - \lambda \sum_{i=1}^{m} \hat{w}_i \log\left(\frac{\hat{w}_i}{w_i}\right) + \eta(1 - \hat{\mathbf{w}}^{\intercal} \mathbf{e}),$$

where λ and η are the Lagrangian variables.

Formulating the Dual Problem. We can now express the dual objective as,

$$g(\lambda, \eta) = \max_{\hat{\mathbf{w}} \geq 0} \mathcal{L}(\hat{\mathbf{w}}, \lambda, \eta), \tag{8}$$

which we need to *minimize*.

Now we borrow the following result from [10],

Lemma 4.1 ([10]). *A configuration Φ is the optimal solution to the* Robust Tuning *problem if and only if* $\min_{\eta,\lambda \geq 0} g(\lambda, \eta) \leq \beta$ *where the minimum is attained for some value of* $\lambda \geq 0$.

In other words, minimizing the dual objective $g(\lambda, \eta)$ (as expressed in Equation (8)) will lead to the optimal solution for the Robust Tuning problem.

Solving the Dual Optimization Problem Optimally. Formulating the dual problem and using the results of Ben-Tal *et al.* [10], we have shown that the dual solution leads to the optimal solution for the Robust Tuning problem. Moreover, we can obtain the optimal solution to the original Robust Tuning problem in polynomial time, a consequence of the tractability of the dual objective.

To solve the dual problem, we first simplify the dual objective $g(\lambda, \eta)$ so that it takes the following form:

$$g(\lambda, \eta) = \eta + \rho\lambda + \lambda \sum_{i=1}^{k} w_i \phi_{KL}^{*}\left(\frac{\mathbf{c}_i(\Phi) - \eta}{\lambda}\right). \tag{9}$$

In Equation (9), $\phi_{KL}^{*}(.)$ denotes the conjugate of KL-divergence function and $\mathbf{c}_i$ corresponds to the i-th dimension of the cost vector $\mathbf{c}(\Phi)$ as defined in Section 3.1 – clearly in this case $k = 4$ as we have 4 types of queries in our workload. Results of Ben-Tal *et al.* [10] show that minimizing the dual function as described in Equation (9) is a convex optimization problem, and it can be solved optimally in polynomial time if and only if the cost function $\mathbf{c}(\Phi)$ is convex in all its dimensions.

In our case, the cost function for the range queries is not convex w.r.t. size ratio T for the tiering policy. However, on account of its smooth non-decreasing form, we are still able to find the global minimum solution for

$$\min_{\Phi,\lambda \geq 0,\eta} \left\{ \eta + \rho\lambda + \lambda \sum_{i=1}^{m} w_i \phi_{KL}^{*}\left(\frac{c_i(\Phi) - \eta}{\lambda}\right) \right\}. \tag{10}$$

This minimization problem can be solved using the Sequential Least Squares Quadratic Programming solver (SLSQP) included in the popular Python optimization library SciPy [78]. Solving this problem outputs the values of the Lagrangian variables λ and η and most importantly the configuration Φ that optimizes the objective of the Robust Tuning problem – for input ρ. In terms of running time, SLSQP solver outputs a robust tuning configuration for a given input in less than a second.

Finding a Value for ρ. Since ρ is a robust tuning parameter, we also provide a few heuristics for setting it. In the presence of historically observed workloads, a DBA may calculate ρ using the following definition: that is, ρ is set to be the largest KL-divergence between any observed workload and the corresponding workload average. If the DBA does not have information about past workloads, they may provide ranges for each query type; then, can sample workloads within those ranges and then calculate ρ using the definiton above to find an appropriate value. DBAs may instead provide two workloads, one that is expected during a normal observation period, and another off-period or unlikely workload. In this case, The KL-divergence between these two workloads can be used as ρ.

5 THE COST MODEL OF LSM TREES

In this section, we provide the detailed cost model used in Endure to accurately capture the behavior of an LSM tree. Following prior work on LSM trees [26, 55], we focus on four types of operations: point queries that return an empty result, point queries that have a match, range queries, and writes.

5.1 Model Basics

When modeling the read cost of LSM trees, a key quantity to accurately capture is the amount of extra read I/Os that take place. While Bloom filters are used to minimize those, they allow for a small fraction of false positives. If the filter returns negative, the target key does not exist in the run, and the lookup skips the access saving one I/O. If a filter returns positive, then the target key may exist, so the lookup probes the run at a cost of one I/O. If the run contains the correct key, the lookup terminates. Otherwise, we have a *false positive* and the lookup continues to probe the next run increasing the I/O cost of a lookup. The false positive rate (ϵ) of a standard Bloom filter designed to query $\mathbf{n}$ entries using a bit-array of size $\mathbf{m}$ is shown by [76] to be calculated as follows:

$$\epsilon = e^{-\frac{\mathbf{m}}{\mathbf{n}} \cdot \ln(2)^2}.$$

Note that the above equation assumes the use of an optimal number of hash functions in the Bloom filter [79].

Classically, LSM tree based key-value stores use the same number of bits-per-entry across all Bloom filters. This means that a lookup probes on average $O\left(e^{-m_{\text{filt}}/N}\right)$ of the runs, where m_{filt} is the overall amount of main memory allocated to the filters. As m_{filt} approaches

0 or infinity, the term $O\left(e^{-m_{\text{filt}}/N}\right)$ approaches 1 or 0 respectively. Here, we build on of the state-of-the-art Bloom filter allocation strategy proposed in Monkey [26] that uses different false positive rates at different levels of the LSM tree to offer optimal memory allocation; for a size ratio T, the false positive rate corresponding to the Bloom filter at the level i is given by

$$f_i(T) = \frac{T^{\frac{T}{T-1}}}{T^{L(T)+1-i}} \cdot e^{-\frac{m_{\text{filt}}}{N}\ln(2)^2}. \tag{11}$$

Additionally, false positive rates for all levels satisfy $0 \leq f_i(T) \leq 1$. It should be further noted that Monkey optimizes false positive rates at individual levels to minimize the worst-case average cost of empty point queries. Non-empty point query costs, being significantly lower than those of empty point queries, are not considered during the optimization process.

LSM Tree Design & System Parameters. In Section 3.1 we introduced the key design and system parameters needed to model LSM tree performance. In addition to those parameters, there are two auxiliary and derived parameters we use in the cost model presented in this section: the potential storage asymmetry in reads and writes (A_{rw}) and the expected selectivity of range queries (S_{RQ}).

5.2 The Cost Model

In this section, we model the costs in terms of expected number of I/O operations required for the fulfillment of the individual queries.

Expected Empty Point Query Cost (Z_0). A point query that returns an empty result will have visited all levels (and every sorted run of every level for tiering), and issues an I/O for every false positive result amongst the Bloom filters. Therefore, the expected number of I/Os per level depends on the Bloom filter memory allocation at that level. Hence, Equation (12) expresses Z_0 in terms of the false positive rates at each level:

$$Z_0(\Phi) = \begin{cases} \sum_{i=1}^{L(T)} f_i(T), & \text{if } \pi = \text{leveling} \\ (T-1)\sum_{i=1}^{L(T)} f_i(T), & \text{if } \pi = \text{tiering.} \end{cases} \tag{12}$$

For leveling, each level has exactly one run, while with tiering, each level has up to $(T-1)$ runs. Each run at the same level in tiering will have equal false positives rates on account of their equal sizes.

Expected Non-empty Point Query Cost (Z). There are two components to the expected non-empty point query cost. First, we assume that the probability of a point query finding a non-empty result in a level is proportional to the size of the level. Thus, the probability of such a query being satisfied on level i by a unit cost I/O operation is simply $\frac{(T-1)\cdot T^{i-1}}{N_f(T)} \cdot \frac{m_{\text{buf}}}{E}$, where $N_f(T)$ denotes the number of entries in a tree completely full up to $L(T)$ levels:

$$N_f(T) = \sum_{i=1}^{L(T)} (T-1)\cdot T^{i-1} \cdot \frac{m_{\text{buf}}}{E}. \tag{13}$$

Second, we assume that all levels preceding level i will trigger an I/O operations with a probability equivalent to the false positive rates of the Bloom filters at those levels. Similarly to empty point queries, the expected cost of such failed I/Os on preceding levels is simply $\sum_{j=1}^{i-1} f_j(T)$. In the case of tiering, we assume that on average, the entry is found in the middle run of the level resulting in an additional $\frac{(T-2)}{2} \cdot f_i(T)$ extra I/Os. Thus, we can compute the non-empty point query cost as an expectation over the entry being found on any of the $L(T)$ levels of the tree as follows:

$$Z_1(\Phi) = \begin{cases} \sum_{i=1}^{L(T)} \frac{(T-1)\cdot T^{i-1}}{N_f(T)} \cdot \frac{m_{\text{buf}}}{E}\left(1 + \sum_{j=1}^{i-1} f_j(T)\right), & \text{if } \pi = \text{leveling} \\ \sum_{i=1}^{L(T)} \frac{(T-1)\cdot T^{i-1}}{N_f(T)} \cdot \frac{m_{\text{buf}}}{E}\left(1 + (T-1)\cdot \sum_{j=1}^{i-1} f_j(T) + \frac{(T-2)}{2}\cdot f_i(T)\right), & \text{if } \pi = \text{tiering.} \end{cases} \tag{14}$$

Range Queries Cost (Q). A range query issues $L(T)$ or $L(T)\cdot(T-1)$ disk seeks (one per run) for leveling and tiering respectively. Each seek is followed by a sequential scan. The cumulative number of pages scanned over all runs is $S_{\text{RQ}} \cdot \frac{N}{B}$, where S_{RQ} is the average proportion of all entries included in range lookups. Hence, the overall range lookup cost Q in terms of pages reads is as follows:

$$Q(\Phi) = \begin{cases} S_{\text{RQ}} \cdot \frac{N}{B} + L(T), & \text{if } \pi = \text{leveling} \\ S_{\text{RQ}} \cdot \frac{N}{B} + L(T)\cdot(T-1), & \text{if } \pi = \text{tiering.} \end{cases} \tag{15}$$

Write Cost (W). We model worst-case writing cost assuming that the vast majority of incoming entries do not overlap. This means that most entries will have to propagate through all levels of the LSM tree. Following the state-of-the-art write cost model, we assume that every written item participated in $\approx \frac{T-1}{T}$ and $\approx \frac{T-1}{2}$ merges with tiering and leveling respectively. We multiply these costs by $L(T)$ since each entry gets merged across all levels, and we divide by the page size B to get the units in terms of I/Os. Since LSM trees often employ solid-state storage that has an asymmetric cost for reads and writes, we represent this storage asymmetry as A_{rw}. For example, a device for which a write operation is twice as expensive as a read operation has $A_{\text{rw}} = 2$. The overall I/O cost is captured by Equation (16):

$$W(\Phi) = \begin{cases} \frac{L(T)}{B} \cdot \frac{(T-1)}{2} \cdot (1 + A_{\text{rw}}), & \text{if } \pi = \text{leveling} \\ \frac{L(T)}{B} \cdot \frac{(T-1)}{T} \cdot (1 + A_{\text{rw}}), & \text{if } \pi = \text{tiering.} \end{cases} \tag{16}$$

When T is set to 2, tiering and leveling behave identically, so the two parts of the equation produce the same result.

Total Expected Cost. The total expected operation cost, $C(\mathbf{w}, \Phi)$, is computed by weighing the empty point lookup cost $Z_0(\Phi)$ from Equation (12), the non-empty point lookup cost $Z(\Phi)$ from Equation (14), the range lookup cost $Q(\Phi)$ from Equation (15), and the write cost $W(\Phi)$ from Equation (16) by their proportion in the workload represented by the terms z_0, z, q and w respectively (Note $z_0 + z_1 + q + w = 1$). This is the exact computation of the cost done in Equation (2) and in the definitions of the Nominal Tuning and Robust Tuning problems (Equations (3) and (4) respectively).

6 UNCERTAINTY BENCHMARK

In this section, we describe the uncertainty benchmark that we use to evaluate the robust tuning configurations given by Endure, both analytically using the cost models, and empirically using RocksDB. It consists of two primary components: (1) *Expected workloads* and, (2) *Benchmark set of sampled workloads*, described below.

Expected Workloads. We create robust tunings configurations for 15 expected workloads encompassing different proportions of

Table 2: Tested expected workloads.

	Expected Workloads														
	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}
z_0:	25%	97%	1%	1%	1%	49%	49%	49%	1%	1%	1%	33%	33%	33%	1%
z_1:	25%	1%	97%	1%	1%	49%	1%	1%	49%	49%	1%	33%	33%	1%	33%
q:	25%	1%	1%	97%	1%	1%	49%	1%	49%	1%	49%	33%	1%	33%	33%
w:	25%	1%	1%	1%	97%	1%	1%	49%	1%	49%	49%	1%	33%	33%	33%
		Unimodal				Bimodal						Trimodal			

query types. We catalog them into *uniform*, *unimodal*, *bimodal*, and *trimodal* categories based upon the dominant query types. While this breakdown of dominant queries is similar to benchmarks such as YCSB, we provide a more comprehensive coverage of potential workloads. A minimum 1% of each query type is always included in every expected workload to ensure a finite KL-divergence. A complete list of all expected workloads is in Table 2.

Benchmark Set of Sampled Workloads. We use the benchmark set of 10K workloads $\mathcal{B}$ as a *test* dataset over which to evaluate the tuning configurations. These configurations are generated as follows: first, we independently sample the number of queries corresponding to each query type uniformly at random from a range $(0, 10000)$ to obtain a 4-tuple of query counts. Next, we divide the individual query counts by the total number of queries in the tuple to obtain a random workload that is added to the benchmark set. We use the actual query counts during the system experimentation where we execute individual queries on the database.

This type of workload breakdown can commonly be seen in LSM trees as shown in a survey of workloads in Facebook's own pipeline [17]. The authors report that ZippyDB, a distributed KV store that uses RocksDB, experiences workloads with 78% gets, 19% writes, and 3% range reads. This breakdown is similar to workload 11, and the exact workload is in the benchmark set $\mathcal{B}$.

Note that while the same $\mathcal{B}$ is used to evaluate different tunings, it represents a different distribution of KL-divergences for the corresponding expected workload associated with each tuning. In the next two sections, we use our uncertainty benchmark to show that tuning with Endure achieves significant performance improvement using both a model-based analysis (Section 7), and an experimental study (Section 8).

7 MODEL-BASED EVALUATION

We now present our detailed model-based study of Endure that uses more than 10000 different noisy workloads for all 15 expected workloads, showing performance benefit of up to 5×. In addition, we show that Endure perfectly matches the classical tuning when there is no uncertainty, that is when the observed workload always matches the expected one, and we pass this information to the robust tuner. Further, we provide recommendations on how to choose uncertainty parameters.

7.1 Evaluation Metrics

Normalized Delta Throughput (Δ). Defining throughput as the reciprocal of the cost of executing a workload, we measure the normalized delta throughput of a configuration Φ_2 w.r.t. another configuration Φ_1 for a given workload $\mathbf{w}$ as follows,

$$\Delta_{\mathbf{w}}(\Phi_1, \Phi_2) = \frac{1/C(\mathbf{w},\Phi_2) - 1/C(\mathbf{w},\Phi_1)}{1/C(\mathbf{w},\Phi_1)}.$$

$\Delta_{\mathbf{w}}(\Phi_1, \Phi_2) > 0$ implies that Φ_2 outperforms Φ_1 when executing a workload $\mathbf{w}$ and vice versa when $\Delta_{\mathbf{w}}(\Phi_1, \Phi_2) < 0$.

Throughput Range (Θ). While normalized delta throughput compares two different tunings, we use the throughput range to evaluate an individual tuning Φ w.r.t. the benchmark set $\mathcal{B}$ as follows,

$$\Theta_{\mathcal{B}}(\Phi) = \max_{\mathbf{w}_0,\mathbf{w}_1 \in \mathcal{B}} \left(\frac{1}{C(\mathbf{w}_0, \Phi)} - \frac{1}{C(\mathbf{w}_1, \Phi)} \right).$$

$\Theta_{\mathcal{B}}(\Phi)$ intuitively captures the best and the worst-case outcomes of the tuning Φ. A smaller value of this metric implies higher consistency in performance.

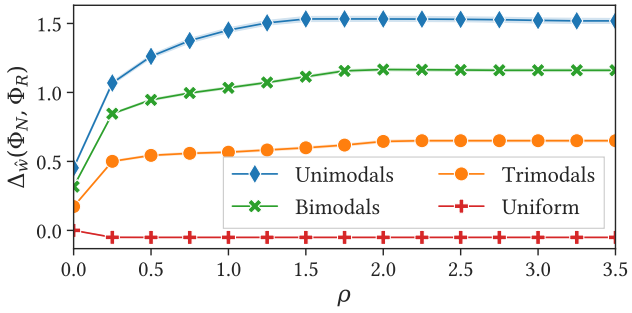


Figure 3: Average delta throughput $\Delta_{\hat{w}}(\Phi_N, \Phi_R)$ for each category of expected workload.

7.2 Experiment Design

To evaluate the performance of our proposed robust tuning approach, we design a large-scale experiment comparing different tunings over the sampled workloads in $\mathcal{B}$ using the analytical cost model. For each of the expected workloads in Table 2, we obtain a single nominal tuning configuration (Φ_N) by solving the Nominal Tuning problem. For 15 different values of ρ in the range (0.0, 4.0) with a step size of 0.25, we obtain a set of robust tuning configurations (Φ_R) by solving the Robust Tuning problem. Finally, we individually compare each of the robust tunings with the nominal over the 10,000 workloads in $\mathcal{B}$ to obtain over 2 million comparisons. While computing the costs, we assume that the database contains 10 million entries each of size 1 KB. The analysis presented in the following sections assumes a total available memory of 10 GB. However, we exhaustively confirmed that changing these parameters does not qualitatively affect the outcomes of our experiment.

7.3 Results

Here, we present an analysis of the comparisons between the robust and the nominal tuning configurations. Using an off-the-shelf global minimizer from the popular Python optimization library SciPy [78], we obtain both nominal and robust tunings with the runtime for the above experiment being less than 10 minutes.

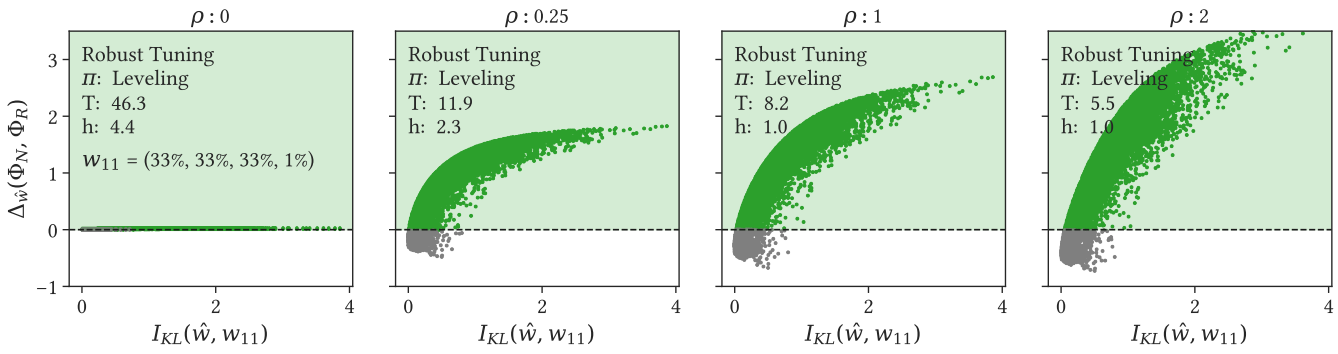
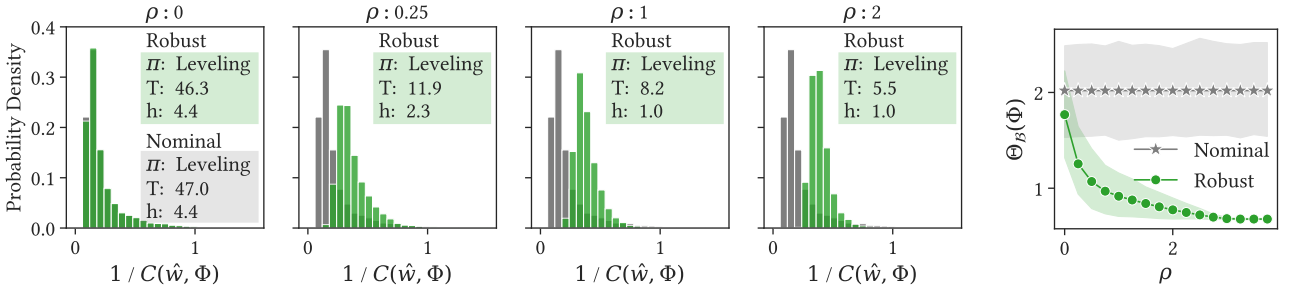


Figure 4: Impact of ρ on normalized delta throughput $\Delta_{\hat{\mathbf{w}}}(\Phi_N, \Phi_R)$ for tunings with expected workload $\mathbf{w}_{11}$.



(a) Histograms of throughput $1/C(\hat{\mathbf{w}}, \Phi)$ for tunings with expected workload $\mathbf{w}_{11}$

(b) Throughput range $\Theta_{\mathcal{B}}(\Phi)$

Figure 5: Impact of ρ on throughput.

Comparison of Tunings. First, we address the question – *is it beneficial to adopt robust tunings relative to the nominal tunings?* Intuitively, it should be clear that the performance of nominally tuned databases would degrade when the workloads being executed on the database are significantly different from the expected workloads used for tuning. In Figure 3, we present performance comparisons between the robust and the nominal tunings for different values of uncertainty parameter ρ. We observe that robust tunings provide substantial benefit in terms of normalized delta throughput for *unimodal*, *bimodal*, and *trimodal* workloads. The normalized delta throughput $\Delta_{\hat{\mathbf{w}}}(\Phi_N, \Phi_R)$ shows over 95% improvement on average over all $\hat{\mathbf{w}} \in \mathcal{B}$ for robust tunings with $\rho \geq 0.5$, when the expected workload used during tuning belongs to one of these categories. For *uniform* expected workload, we observe that the nominal tuning outperforms the robust tuning by a modest 5%.

Intuitively, *unbalanced* workloads result in overfit nominal tunings. Hence, even small variations in the observed workload can lead to significant degradation in the throughput of such nominally tuned databases. On the other hand, robust tunings by their very nature take into account such variations and comprehensively outperform the nominal tunings. In the case of the uniform expected workload $\mathbf{w}_0$, a low value of ρ covers a larger area of possible workloads then that same value would in a different workload as evident in Figure 2. In this case, when tuned for high values of ρ, the robust tunings are unrealistically pessimistic and lose out some performance relative to the nominal tuning.

Impact of Tuning Parameter ρ. Next, we address the question – *how does the uncertainty tuning parameter ρ impact the performance of the robust tunings?* In Figure 4, we take a deep dive into the performance of robust tunings for an individual expected workload for different values of ρ. We observe that the robust tunings for $\rho = 0$ i.e., zero uncertainty, are very close to the nominal tunings. As the value of ρ increases, its performance advantage over the nominal tuning for the observed workloads with higher KL-divergence w.r.t. expected workload increases. Furthermore, the robustness of such configurations have logically sound explanations. The expected workload in Figure 4 consists of just 1% writes. Hence, for low values of ρ, the robust tuning has higher size ratio leading to shallower LSM trees to achieve good read performance. For higher values of ρ, the robust tunings anticipate an increasing percentage of write queries and hence limit the size ratio to achieve higher throughput.

In Figure 5, we show the impact of tuning parameter ρ on the throughput range. In Figure 5a we plot a histogram of the nominal and robust throughputs for workload $\mathbf{w}_{11}$. As the value of ρ increases, the interval size between the lowest and the highest throughputs for the robust tunings consistently decreases. We provide further evidence of this phenomenon in Figure 5b, by plotting the decreasing throughput range $\Theta_{\mathcal{B}}(\Phi_R)$ averaged across all the expected workloads. Thus, robust tunings not only provide a higher average throughput over all $\hat{\mathbf{w}} \in \mathcal{B}$, but they have a more consistent performance (lower variance) compared to the nominal tunings.

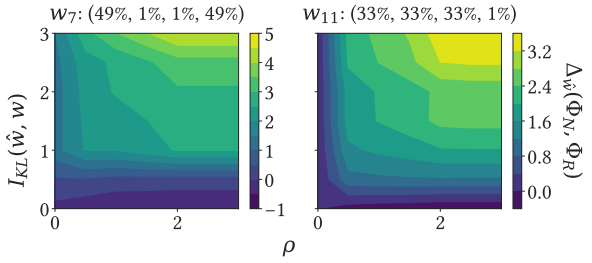


Figure 6: Delta throughputs $\Delta_{\hat{\mathbf{w}}}(\Phi_N, \Phi_R)$ for ρ vs $I_{KL}(\hat{\mathbf{w}}, \mathbf{w})$.

Choice of ρ. Now, we address the question – *What is the appropriate choice for the value of uncertainty parameter ρ?* In Figure 6, we explore the relationship between ρ and the KL-divergence $I_{KL}(\hat{\mathbf{w}}, \mathbf{w})$ for $\hat{\mathbf{w}} \in \mathcal{B}$, by making a contour plot of the corresponding normalized delta throughput $\Delta_{\hat{\mathbf{w}}}(\Phi_N, \Phi_R)$. We confirm our intuition that nominal tunings compare favorably with our proposed robust tunings only in two scenarios: (1) when the observed workloads are extremely similar to the expected workload (close to zero observed uncertainty), and (2) when the robust tunings assume extremely low uncertainty with $\rho < 0.2$ while the observed variation is higher. Based on this evidence, we propose the following rule of thumb: the maximum KL-divergence between any two pairs of observed workloads is a reasonable value of ρ in practice.

8 SYSTEM-BASED EVALUATION

In this section, we deploy Endure as the tuner of the state-of-the-art LSM-based engine RocksDB, and we show that RocksDB achieves up to 90% lower workload latency in the presence of uncertainty. We further show that the tuning cost is negligible, and the effectiveness of Endure is not affected by data size.

8.1 Experimental Setup & Measurements

Our server is configured with two Intel Xeon Gold 6230 processors, 384 GB of main memory, a 1 TB Dell P4510 NVMe drive, CentOS 7.9.2009, and a default page size of 4 KB. We use Facebook's RocksDB, a popular LSM tree-based storage system, to evaluate our approach [31]. While RocksDB provides implementations of leveling and tiering policies, the system implements micro-optimizations not common across all LSM tree-based storage engines. Therefore, we use RocksDB's event hooks to implement both classic leveling and tiering policies to benchmark the common compaction strategies. Following the Monkey memory allocation scheme [26], we allocate different bits per element for Bloom filters per level. To obtain an accurate count of block accesses we enable direct I/Os for both queries and compaction and disable the block cache. The remaining parameters such as buffer size are set by the tuning.

Empirical Measurements. We use the internal RocksDB statistics module to measure the number of logical block accesses during reads, bytes flushed during writes, and bytes read and written in compactions. The number of logical blocks accessed during writes is calculated by dividing the number of bytes reported by the default page size. To estimate the amortized cost of writes, we compute the I/Os from compactions across all workloads of a session and redistribute them across write queries. Our approach of measuring average I/Os per query allows us to compare the effects of different tuning configurations, while simultaneously minimizing the effects of extraneous factors on the database performance.

8.2 Experiment Design

To evaluate the performance of our proposed robust tuning approach, we create multiple instances of RocksDB using different tunings and empirically measure their performance by executing workloads from the uncertainty benchmark $\mathcal{B}$. To measure the steady-state performance of the database, each instantiation is initially bulk loaded with the exact same sequence of 10 million unique key-value pairs each of size 1 KB. Each key-value entry has a 16-bit uniformly at random sampled key, with the remaining bits being allocated to a randomly generated value.

While evaluating the performance of the database, we sample a sequence of workloads from the benchmark set $\mathcal{B}$. For each workload, the observation period is set to 200,000 queries, therefore each experiment issues a total of 6 million queries to the database. Each sequence is cataloged into one of the categories — *expected*, *empty read*, *non-empty read*, *read*, *range*, and *write* — based on the dominant query type in the workloads in the sequence. Specifically, the *expected* session contains workloads with a KL-divergence less than 0.2 w.r.t. the expected workload used for tuning. In all other sessions, the dominant query type encompasses 80% of the total queries in the session. The remaining 20% of queries may belong to any of the query types. While generating keys of the queries to run on the database, we ensure that non-empty point reads query a key that exists in the database, while the empty point reads query a key that is not present in the database but is sampled from the same domain. All range queries are generated with minimal selectivity S_{RQ} to act as short range queries reading on average zero to two pages per level. Lastly, write queries consist of randomly generated keys that are distinct from the existing keys in the database. Initializing RocksDB and bulk loading requires 30 mins; the execution of individual workloads takes on average 5 mins.

8.3 Experimental Results

In this section, we replicate key insights from Section 7, evaluate system performance, and show that Endure scales with database size. Due to space constraints, we present detailed results for expected workloads, $\mathbf{w}_7$ and $\mathbf{w}_{11}$. However, Table 3 summarizes the normalized delta throughputs $\Delta_{\mathbf{w}}(\Phi_N, \Phi_R)$ for each expected workload in $\mathcal{B}$. The interested reader can find detailed plots for all workloads in the extended version of this work [41].

Cost of Tuning. For every experiment, we solve for either the nominal or the robust tuning prior to the experiments. Solving for nominal and robust tuning takes less than 10ms, which is negligible w.r.t. workload execution time.

Read Performance. We begin by examining the system performance and verifying that the model-predicted I/O and the system-measured I/O match when considering read queries in Figures 7 and 8. In both figures, we include the model-predicted I/Os per query (top), the I/Os per query measured on the system (middle), and the system latency (bottom), for both nominal and robust tunings across different read sessions. Additionally, the total throughput numbers in queries per second are reported at the end of the

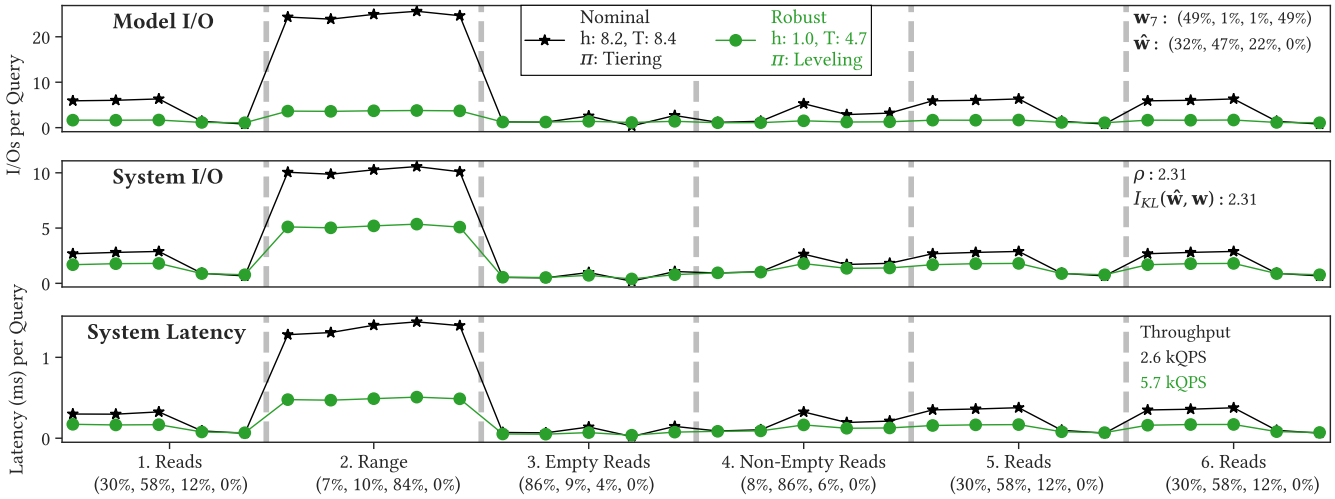


Figure 7: System and model performance for robust and nominal tunings in a read-only query sequence. Here the tuning parameter ρ matches the observed value of $I_{KL}(\hat{\mathbf{w}}, \mathbf{w}_7)$. Each session contains the label and average workload.

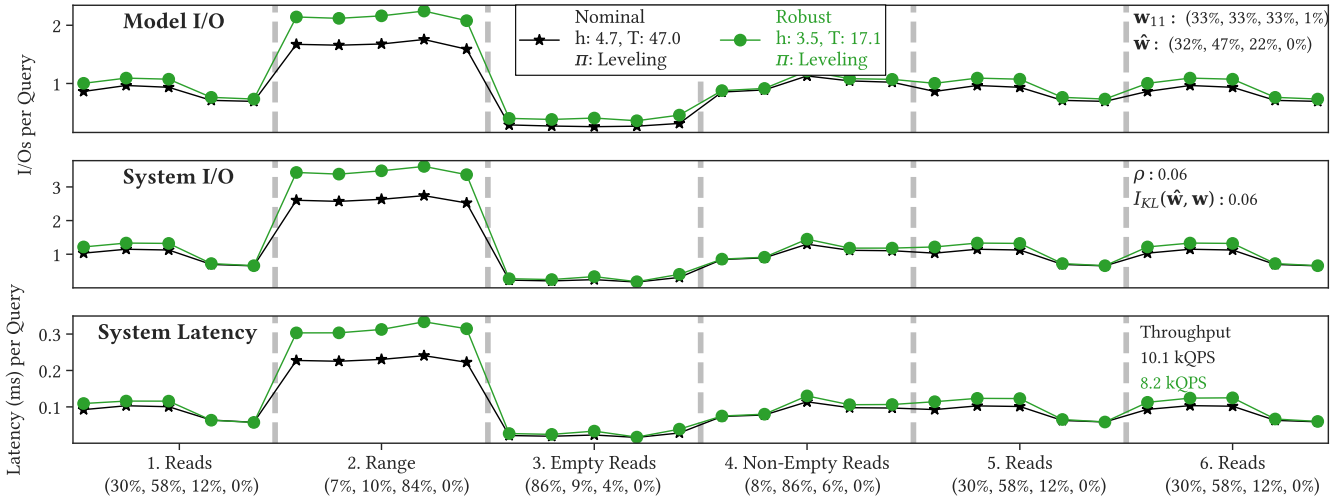


Figure 8: Read-only sequence where the observed workloads $\hat{\mathbf{w}}$ is close to the expected, hence ρ and $I_{KL}(\hat{\mathbf{w}}, \mathbf{w}_{11})$ deviate.

system latency graph. The empirical measurements confirm that the predicted performance benefits from the model translate to similar performance benefits in practice. Note that the discrepancy observed between the relative performance between the nominal and the robust tunings in the presence of range queries (session 2 in Figure 7) is due to the fence pointers in RocksDB. The analytical model does not account for fence pointers allowing the system to completely skip a run, which may reduce the measured I/Os for short range queries compared to the predicted I/Os.

Write Performance. In presence of writes in Figure 9, the model is still predicting the disk accesses successfully and Endure leads to significant performance benefits. Note that now the structure of the LSM tree is continually changing across all sessions due to the compactions caused by write queries. For example, in Figure 9 the dip in measured I/Os and latency in the range-query session are the result of empty levels being created via compactions triggered from preceding workloads. From the write session of Figure 9, we observe that the nominal tuning suffers from high latency and I/O cost. This is due to the large size ratio T that creates a shallow tree with extremely large levels which suffers from a long stalls during compactions. Compare this to the robust tuning: the lower size ratio creates a tree with more stable performance for both I/Os per query and query latency which leads to a higher overall throughput. Overall, we observe that the robust tuning reduces I/O and latency by up to 90%. Figures 7–9 confirm that our analytical model can accurately capture the relative performance of different tunings.

Robust Outperforms Nominal for Properly Selected ρ. In the model evaluation (Figure 6), we showed that robust tuning outperforms the nominal tuning in the presence of uncertainty for tuning parameter ρ approximately greater than 0.2. This is further supported by all the system experiments described. Specifically, Figures 7 and 9 show instances where the KL-divergence of the

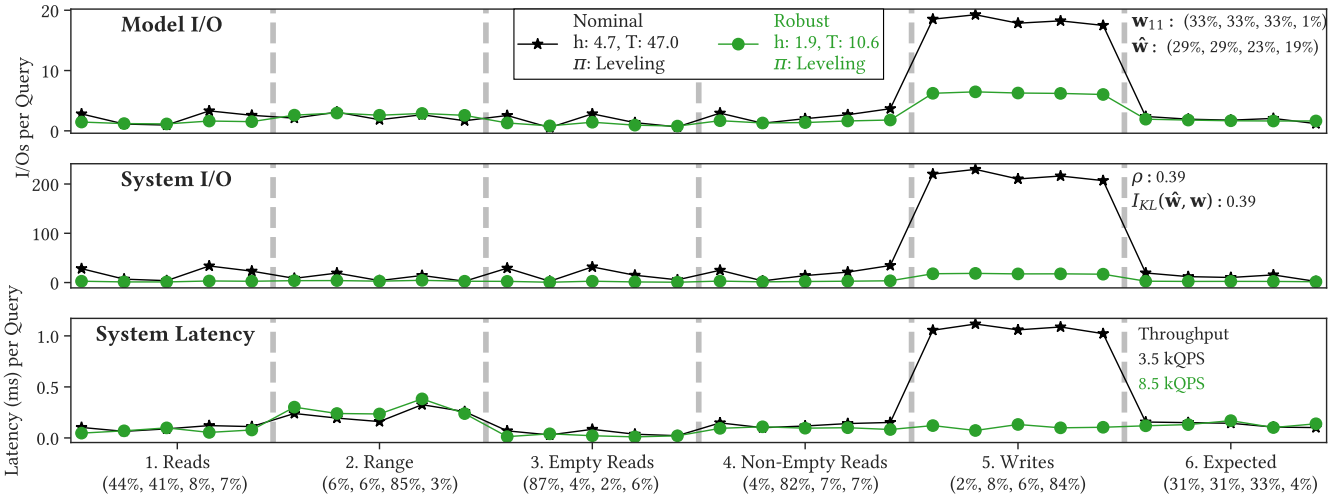


Figure 9: Sequence where ρ and $I_{KL}(\hat{w}, w_{11})$ closely match. Both system I/O and latency show reductions of up to 90%.

Table 3: System measured normalized delta throughputs $\Delta_w(\Phi_N, \Phi_N)$ and their respective tunings for experiments on all expected workloads in $\mathcal{B}$ with an optimally selected ρ.

Expected	$\Phi = (T, m_{filt}, \pi)$		
Workload (w)	Φ_N	Φ_R	$\Delta_w(\Phi_N, \Phi_R)$
w_0	(5.2, 3.5, L)	(5.1, 3.1, L)	0.0
w_1	(5.7, 9.4, L)	(5.0, 4.2, L)	0.0
w_2	(5.8, 5.3, L)	(5.0, 1.0, L)	0.1
w_3	(100, 0.0, L)	(5.4, 1.0, L)	0.4
w_4	(17, 3.2, T)	(4.6, 1.0, L)	1.5
w_5	(5.5, 8.8, L)	(5.1, 3.9, L)	0.1
w_6	(63, 4.8, L)	(8.2, 1.0, L)	0.8
w_7	(8.4, 8.2, T)	(3.4, 1.0, L)	0.5
w_8	(62, 0.0, L)	(8.0, 1.0, L)	0.6
w_9	(8.3, 6.9, T)	(3.3, 1.0, L)	0.8
w_{10}	(5.0, 0.0, L)	(5.0, 1.0, L)	0.0
w_{11}	(47, 4.7, L)	(11, 1.9, L)	1.4
w_{12}	(6.2, 8.1, T)	(2.8, 3.1, L)	0.2
w_{13}	(5.1, 3.5, L)	(5.0, 1.0, L)	-0.1
w_{14}	(5.1, 0.0, L)	(5.0, 1.0, L)	-0.1

observed workload averaged across all the sessions w.r.t. the expected workload is close to the tuning parameter ρ. Additionally, we present results for all expected workloads in Table 3. Each entry in the table summarizes the total throughput after running the same experimental setup presented in Figure 9. We observe that the robust outperforms the nominal in 10 of our expected workloads, with only 2 workloads where robust tuning does worse, however, in these cases the reported throughputs are comparable. In each of these experiments, the robust tuning outperforms the nominal resulting in up to a 90% reduction in latency and system I/O. Lastly, in Figure 8, the observed workloads are similar to the expected one ($I_{KL}(\hat{w}, w_{11}) < 0.2$), resulting in a latency increase of 20%.

ENDURE Scales with Data Size. To verify that ENDURE scales, we repeat the previous experiments, while varying the size of the initial

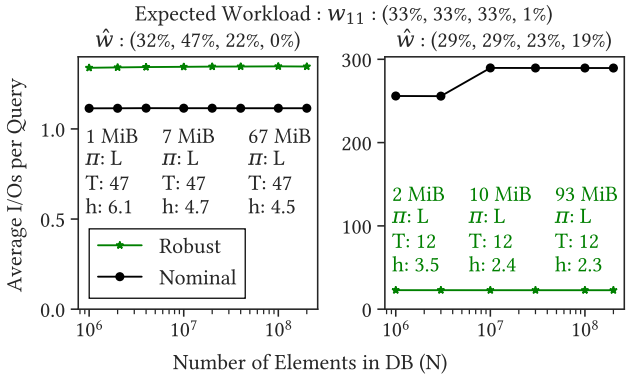


Figure 10: Impact of database size on performance. All tunings use the same expected workload w_{11} with executed workloads shown above each graph. Points at each power of 10 show m_{buf} and the tuning Φ (L for leveling, T for tiering).

database. Each point in Figure 10 is calculated based on a series of workload sessions similar to the ones presented in Figures 8 (9) for the left (right) part of Figure 10. All points use the same expected workload, therefore the nominal and robust tunings are the same across each graph. We observe that the robust and nominal tuning increase buffer memory as the initial database size grows. As a result, for all cases, the number of initial levels is the same regardless of the number of entries. This highlights the importance of the number of levels w.r.t performance. Finally, the performance gap between robust and nominal stays consistent as database size grows, showing that ENDURE is effective regardless of data size.

8.4 Robustness is All You Need

One of the key challenges during the evaluation of tuning configurations in presence of uncertainty is the challenge in measuring steady-state performance. In Section 8.3, we show that the cost-model can accurately predict the empirical measurements. In the

course of this study, using our model, we compared over 700 different robust tunings with their nominal counterparts over the uncertainty benchmark set $\mathcal{B}$, leading to approximately 8.6 million comparisons. Robust tunings comprehensively outperform the nominal tunings in over 80% of these comparisons. We further cross-validated the relative performance of the nominal and the robust tunings in over 300 comparisons using RocksDB. The empirical measurements overwhelmingly confirmed the validity of our analytical models, and the few instances of discrepancy in the scale of measured I/Os, such as the ones discussed in previous sections, are easily explained based on the structure of the LSM tree.

Leveling is "more" Robust than Tiering. One of the key takeaways of applying robust tuning to LSM trees is that *leveling is inherently more robust* to perturbations in workloads when compared to pure tiering. Note that this is evident from Table 3, where all robust tunings suggest leveling as the compaction policy. This observation is in line with the industry practice of deploying leveling or hybrid leveling over pure tiering. Overall, based on our analytical and empirical results, the robust tuning should always be employed when tuning an LSM tree, unless the future workload distribution is known with absolute certainty.

Discussion. While we have deployed and tested robust tuning on LSM trees, the robust paradigm of Endure is a generalization of a minimization problem that is at the heart of any database tuning problem. Hence, similar robust optimization approaches can be applied to *any database tuning* problem assuming that the underlying cost-model is known, and each cost-model component is convex or can be accurately approximated by a convex surrogate.

9 RELATED WORK

Tuning Data Systems. Database systems are notorious for having numerous tuning knobs. These tuning knobs control fine-grained decisions (e.g., number of threads, amount of memory for bufferpool, storage size for logging) as well as basic architectural and physical design decisions about partitioning, index design, materialized views that affect storage and access patterns, and query execution [15, 21]. The database research community has developed several tools to deal with such tuning problems. These tools can be broadly classified as offline workload analysis for index and views design [2, 3, 20, 24, 77, 85], and periodic online workload analysis [16, 68–70] to capture workload drift [39]. In addition, there has been research on reducing the magnitude of the search space of tuning [15, 25] and on deciding the optional data partitioning [9, 61, 72, 74, 75]. These approaches assume that the input information about resources and workload is accurate. When it is proved to be inaccurate, performance is typically severely impacted.

Adaptive & Self-designing Data Systems. A first attempt to address this problem was the design of *adaptive* systems which had to pay additional transition costs (e.g., when deploying a new tuning) to accommodate shifting workloads [34, 35, 43, 71]. More recently the research community has focused on using machine learning to learn the interplay of tuning knobs, and especially of the knobs that are hard to analytically model to perform cost-based optimization. This recent work on self-driving database systems [4, 56, 62] or self-designing database systems [42, 44–46] is exploiting new advancements in machine learning to tune database systems and reduce the need for human intervention, however, they also yield suboptimal results when the workload and resource availability information is inaccurate.

Robust Database Physical Design. One of the key database tuning decisions is physical design, that is, the decision of which set of auxiliary structures should be used to allow for the fastest execution of future queries. Most of the existing systems use past workload information as a representative sample for future workloads, which often leads to sub-optimal decisions when there is significant workload drift. Cliffguard [58] is the first attempt to use unconstrained robust optimization to find a robust physical design. Their method is derived from Bertsimas et al. in [12], a numerical optimization approach using alternating gradient ascent-descent to optimize problems without closed-form objectives. In contrast to Cliffguard, Endure focuses on the LSM tree tuning problem which uses an analytical closed form objective in Equation (2). This allows us to directly solve a Lagrangian dual problem instead of relying upon numerical optimization techniques. Furthermore, we found that the approach in Cliffguard, when applied to our objective, fails to converge even after extensive hyperparameter search.

10 CONCLUSION

In this work, we explored the impact of workload uncertainty on the performance of LSM tree-based databases and introduced Endure, a robust tuning paradigm that recommends optimal designs to mitigate any performance degradation in the presence of workload uncertainty. We showed that in the presence of uncertainty, robust tunings increase database throughput compared to standard tunings by up to 5×. Additionally, we provided evidence that our analytical model closely matches the behavior measured on a database system. Through both model-based and extensive experimental evaluation of Endure within the state-of-the-art LSM-based storage engine, RocksDB, we show that the robust tuning methodology consistently outperforms classical tuning strategies. Endure can be an indispensable tool for database administrators to evaluate deployed tunings' performance as well as recommend optimal tunings in a few seconds without resorting to expensive database experiments.

ACKNOWLEDGEMENTS

We would like to thank Josh Berkus for his valuable feedback. This work is partially funded by IBM Ph.D. Fellowship Award, RedHat Incubation Award, NSF Grants – No. III-1813406, No. IIS-1850202, No. III-1908510, and No. IIS-2144547.

REFERENCES

[1] Ildar Absalyamov, Michael J Carey, and Vassilis J Tsotras. 2018. Lightweight Cardinality Estimation in LSM-based Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 841–855. https://doi.org/10.1145/3183713.3183761

[2] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. 2004. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 1110–1121.

[3] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 496–505. http://dl.acm.org/citation.cfm?id=645926.671701

[4] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1009–1024. https://doi.org/10.1145/3035918.3064029

[5] Wail Y Alkowaileet, Sattam Alsubaiee, and Michael J Carey. 2020. An LSM-based Tuple Compaction Framework for Apache AsterixDB. *Proceedings of the VLDB Endowment* 13, 9 (2020), 1388–1400. http://www.vldb.org/pvldb/vol13/p1388-alkowaileet.pdf

[6] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1905–1916. https://doi.org/10.14778/2733085.2733096

[7] Andy Huynh, Harshal A. Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2021. Robust LSM Tuning. https://github.com/BU-DiSC/endure

[8] Apache. 2021. Cassandra. *http://cassandra.apache.org* (2021).

[9] Manos Athanassoulis, Kenneth S. Bøgh, and Stratos Idreos. 2019. Optimal Column Layout for Hybrid Workloads. *Proceedings of the VLDB Endowment* 12, 13 (2019), 2393–2407.

[10] Aharon Ben-Tal, Dick den Hertog, Anja De Waegenaere, Bertrand Melenberg, and Gijs Rennen. 2013. Robust Solutions of Optimization Problems Affected by Uncertain Probabilities. *Management Science* 59, 2 (2013), 341–357. https://doi.org/10.1287/mnsc.1120.1641

[11] Aharon Ben-Tal and Arkadi Nemirovski. 1998. Robust Convex Optimization. *Mathematics of Operations Research* 23, 4 (1998), 769–805. https://doi.org/10.1287/moor.23.4.769

[12] Dimitris Bertsimas, Omid Nohadani, and Kwong Meng Teo. 2010. Robust Optimization for Unconstrained Simulation-Based Problems. *Operations Research* 58, 1 (2010), 161–178. https://doi.org/10.1287/opre.1090.0715

[13] Burton H Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426. http://dl.acm.org/citation.cfm?id=362686.362692

[14] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. 2018. Accordion: Better Memory Organization for LSM Key-Value Stores. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1863–1875. http://www.vldb.org/pvldb/vol11/p1863-bortnikov.pdf

[15] Nicolas Bruno and Surajit Chaudhuri. 2005. Automatic physical database tuning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 227–238. https://doi.org/10.1145/1066157.1066184

[16] Nicolas Bruno and Surajit Chaudhuri. 2006. To Tune or not to Tune? A Lightweight Physical Design Alerter. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 499–510. http://dl.acm.org/citation.cfm?id=1182635.1164171

[17] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H C Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 209–223.

[18] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 205–218. http://dl.acm.org/citation.cfm?id=1267308.1267323

[19] Surajit Chaudhuri, Benoit Dageville, and Guy M Lohman. 2004. Self-Managing Technology in Database Management Systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 1243. https://doi.org/10.1016/B978-012088469-8.50116-9

[20] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 146–155. http://dl.acm.org/citation.cfm?id=645923.673646

[21] Surajit Chaudhuri and Vivek R Narasayya. 1998. AutoAdmin 'What-if' Index Analysis Utility. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 367–378. https://doi.org/10.1145/276304.276337

[22] Surajit Chaudhuri and Gerhard Weikum. 2005. Foundations of automated database tuning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 964–965. https://doi.org/10.1145/1066157.1066305

[23] Navraj Chohan, Claris Castillo, Mike Spreitzer, Malgorzata Steinder, Asser N Tantawi, and Chandra Krintz. 2010. See Spot Run: Using Spot Instances for MapReduce Workflows. In *Proceedings of USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.

[24] Benoit Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. 2004. Automatic SQL tuning in oracle 10g. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 1098–1109. http://dl.acm.org/citation.cfm?id=1316689.1316784

[25] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. 2011. CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads. *Proceedings of the VLDB Endowment* 4, 6 (2011), 362–372. https://doi.org/10.14778/1978665.1978668

[26] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 79–94. https://doi.org/10.1145/3035918.3064054

[27] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 505–520. https://doi.org/10.1145/3183713.3196927

[28] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 449–466. https://doi.org/10.1145/3299869.3319903

[29] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 205–220. https://doi.org/10.1145/1323293.1294281

[30] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf

[31] Facebook. 2021. RocksDB. *https://github.com/facebook/rocksdb* (2021).

[32] Guilherme Galante and Luis Carlos Erpen De Bona. 2012. A Survey on Cloud Computing Elasticity. In *Proceedings of the IEEE International Conference on Utility and Cloud Computing (UCC)*. 263–270. https://doi.org/10.1109/UCC.2012.30

[33] Google. 2021. LevelDB. *https://github.com/google/leveldb/* (2021).

[34] Goetz Graefe and Harumi Kuno. 2010. Self-selecting, self-tuning, incrementally optimized indexes. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 371–381. http://dl.acm.org/citation.cfm?id=1739041.1739087

[35] Goetz Graefe and Harumi A. Kuno. 2010. Adaptive indexing for relational keys. In *Proceedings of the IEEE International Conference on Data Engineering Workshops (ICDEW)*. 69–74.

[36] Brian Hayes. 2008. Cloud computing. *Commun. ACM* 51, 7 (2008), 9–11. https://doi.org/10.1145/1364782.1364786

[37] HBase. 2013. Online reference. *http://hbase.apache.org/* (2013).

[38] Nikolas Roman Herbst, Samuel Kounev, and Ralf H Reussner. 2013. Elasticity in Cloud Computing: What It Is, and What It Is Not. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*. 23–27.

[39] Marc Holze, Ali Haschimi, and Norbert Ritter. 2010. Towards workload-aware self-management: Predicting significant workload shifts. *Proceedings of the IEEE International Conference on Data Engineering (ICDE)* (2010), 111–116. https://doi.org/10.1109/ICDEW.2010.5452738

[40] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 651–665. https://doi.org/10.1145/3299869.3314041

[41] Andy Huynh, Harshal A Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2021. Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty. *CoRR* 2110.13801 (2021). https://arxiv.org/abs/2110.13801

[42] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, and Zichen Zhu. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

[43] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

[44] Stratos Idreos and Tim Kraska. 2019. From Auto-tuning One Size Fits All to Self-designed and Learned Data-intensive Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[45] Stratos Idreos, Kostas Zoumpatianos, Manos Athanassoulis, Niv Dayan, Brian Hentschel, Michael S. Kester, Demi Guo, Lukas M. Maas, Wilson Qin, Abdul Wasay, and Yiyou Sun. 2018. The Periodic Table of Data Structures. *IEEE*

Data Engineering Bulletin 41, 3 (2018), 64–75. http://sites.computer.org/debull/A18sept/p64.pdf

[46] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S Kester, and Demi Guo. 2018. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 535–550. https://doi.org/10.1145/3183713.3199671

[47] Influxdata. 2021. In-memory indexing and the Time-Structured Merge Tree (TSM). *https://docs.influxdata.com/influxdb/v1.8/concepts/storage_engine/* (2021).

[48] Intel. 2011. Best Practices for Building an Enterprise Private Cloud. *White Paper* (2011).

[49] Taewoo Kim, Alexander Behm, Michael Blow, Vinayak Borkar, Yingyi Bu, Michael J. Carey, Murtadha Hubail, Shiva Jahangiri, Jianfeng Jia, Chen Li, Chen Luo, Ian Maxon, and Pouria Pirzadeh. 2020. Robust and efficient memory management in Apache AsterixDB. *Software - Practice and Experience* 50, 7 (2020), 1114–1151. https://doi.org/10.1002/spe.2799

[50] Solomon Kullback and Richard A. Leibler. 1951. On Information and Sufficiency. *The Annals of Mathematical Statistics* 22, 1 (1951), 79–86. https://doi.org/10.1214/aoms/1177729694

[51] Chen Luo. 2020. Breaking Down Memory Walls in LSM-based Storage Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2817–2819. https://doi.org/10.1145/3318464.3384399

[52] Chen Luo and Michael J Carey. 2019. On Performance Stability in LSM-based Storage Systems. *Proceedings of the VLDB Endowment* 13, 4 (2019), 449–462.

[53] Chen Luo and Michael J. Carey. 2020. LSM-based Storage Techniques: A Survey. *The VLDB Journal* 29, 1 (2020), 393–418. https://doi.org/10.1007/s00778-019-00555-y

[54] Chen Luo, Pinar Tözün, Yuanyuan Tian, Ronald Barber, Vijayshankar Raman, and Richard Sidle. 2019. Umzi: Unified Multi-Zone Indexing for Large-Scale HTAP. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 1–12. https://doi.org/10.5441/002/edbt.2019.02

[55] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2071–2086. https://doi.org/10.1145/3318464.3389731

[56] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 631–645. https://doi.org/10.1145/3183713.3196908

[57] C Mohan. 2016. Hybrid Transaction and Analytics Processing (HTAP): State of the Art. In *Proceedings of the International Workshop on Business Intelligence for the Real-Time Enterprise (BIRTE)*.

[58] Barzan Mozafari, Eugene Zhen Ye Goh, and Dong Young Yoon. 2015. CliffGuard: A Principled Framework for Finding Robust Database Designs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1167–1182. https://doi.org/10.1145/2723372.2749454

[59] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385. http://dl.acm.org/citation.cfm?id=230823.230826

[60] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid Transactional/Analytical Processing: A Survey. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1771–1775. https://doi.org/10.1145/3035918.3054784

[61] Stratos Papadomanolakis and Anastasia Ailamaki. 2004. AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*. 383. https://doi.org/10.1109/SSDBM.2004.19

[62] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. http://cidrdb.org/cidr2017/papers/p42-pavlo-cidr17.pdf

[63] Massimo Pezzini, Donald Feinberg, Nigel Rayner, and Roxane Edjlali. 2014. Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation. *https://www.gartner.com/doc/2657815/* (2014). https://www.gartner.com/doc/2657815/

[64] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2037–2048. http://www.vldb.org/pvldb/vol10/p2037-ren.pdf

[65] Grand View Research. 2019. Private Cloud Server Market Size, Share & Trend Analysis Report By Hosting Type (User Hosting, Provider Hosting), By Organization Type (SME, Large Enterprise), By Region, And Segment Forecasts, 2019 - 2025. *White Paper* (2019).

[66] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A Tunable Delete-Aware LSM Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 893–908. https://doi.org/10.1145/3318464.3389757

[67] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2216–2229. http://vldb.org/pvldb/vol14/p2216-sarkar.pdf

[68] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. 2006. COLT: Continuous On-Line Database Tuning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 793–795. https://doi.org/10.1145/1142473.1142592

[69] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. 2007. On-Line Index Selection for Shifting Workloads. In *Proceedings of the IEEE International Conference on Data Engineering Workshops (ICDEW)*. 459–468. https://doi.org/10.1109/ICDEW.2007.4401029

[70] Karl Schnaitter and Neoklis Polyzotis. 2012. Semi-automatic index tuning. *Proceedings of the VLDB Endowment* 5, 5 (2012), 478–489. https://doi.org/10.14778/2140436.2140444

[71] Felix Martin Schuhknecht, Jens Dittrich, and Laurent Linden. 2018. Adaptive Adaptive Indexing. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 665–676. https://doi.org/10.1109/ICDE.2018.00066

[72] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-Grained Adaptive Partitioning for General Database Schemas. *Proceedings of the VLDB Endowment* 10, 4 (2016), 445–456. http://www.vldb.org/pvldb/vol10/p445-serafini.pdf

[73] Dennis E Shasha and Philippe Bonnet. 2002. Database Tuning: Principles, Experiments, and Troubleshooting Techniques. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.

[74] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. 2014. Fine-grained Partitioning for Aggressive Data Skipping. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1115–1126. https://doi.org/10.1145/2588555.2610515

[75] Liwen Sun, Michael J. Franklin, Jiannan Wang, and Eugene Wu. 2016. Skipping-oriented Partitioning for Columnar Layouts. *Proceedings of the VLDB Endowment* 10, 4 (2016), 421–432. http://www.vldb.org/pvldb/vol10/p421-sun.pdf

[76] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. 2012. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Communications Surveys & Tutorials* 14, 1 (2012), 131–155. http://ieeexplore.ieee.org/xpl/login.jsp?arnumber=5751342

[77] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend its own Indexes. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 101–110. http://dx.doi.org/10.1109/ICDE.2000.839397

[78] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, and et al. Cournapeau D. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[79] Wikipedia contributors. 2021. Bloom filter – Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Bloom_filter. [Online; accessed 8-June-2021].

[80] WiredTiger. 2021. Source Code. *https://github.com/wiredtiger/wiredtiger* (2021).

[81] Rich Wolski, John Brevik, Ryan Chard, and Kyle Chard. 2017. Probabilistic guarantees of execution duration for Amazon spot instances. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 18:1—-18:11. https://doi.org/10.1145/3126908.3126953

[82] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. 2020. Leaper: A Learned Prefetcher for Cache Invalidation in LSM-tree based Storage Engines. *Proceedings of the VLDB Endowment* 13, 11 (2020), 1976–1989.

[83] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 323–336. https://doi.org/10.1145/3183713.3196931

[84] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, Zhongdong Huang, and Jianling Sun. 2020. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 225–237.

[85] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. 2004. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 1087–1097. http://dl.acm.org/citation.cfm?id=1316689.1316783