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

Anonymous Author(s)

ABSTRACT

Log-structured merge-trees (LSM trees) are increasingly used as the storage engines behind several data systems, many of which are deployed in the cloud. Similar to other database architectures, LSM trees take into account information about the *expected* workloads (e.g., reads vs. writes and point vs. range queries) and optimize their performances by changing tunings. Operating in the cloud, however, comes with a degree of *uncertainty* due to multi-tenancy and the fast-evolving nature of modern applications. Databases with static tunings 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 query 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 an expected workload. Additionally, an uncertainty tuning parameter controls the size of this neighborhood, thereby allowing the output tunings to be conservative or optimistic. We benchmark Endure on a state-of-the-art LSM-based storage engine, RocksDB, and show that its tunings comprehensively outperform tunings from classical strategies. Drawing upon the results of our extensive analytical and empirical evaluation, we recommend the use of Endure for optimizing the performance of LSM tree-based storage engines.

ACM Reference Format:

Anonymous Author(s). 2021. Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Ubiquitous LSM-based Key-Value Stores. Log-Structured Merge trees (LSM trees) is the most commonly deployed data structure used in the backend storage of modern key-value stores [58]. LSM trees offer high ingestion rate and fast reads, making them widely adopted by systems such as RocksDB [32] at Facebook, LevelDB [34] and BigTable [17] at Google, HBase [38] and Cassandra [7] at Apache, WiredTiger [81] at MongoDB, X-Engine [41] at Alibaba, and DynamoDB [29] at Amazon.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

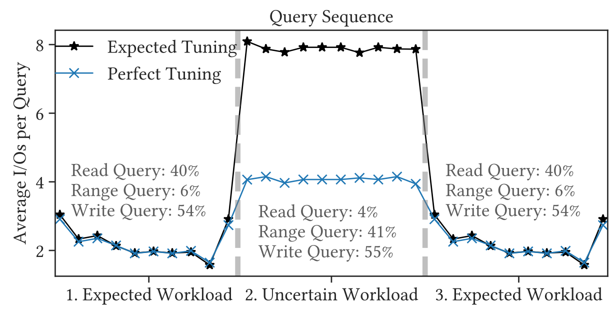


Figure 1: LSM tree tunings and their performance on observed workloads. While both workloads have a similar ratio of reads and writes, the uncertain workload has a higher percentage of range queries leading to the expected system tuning to experience a $2\times$ degradation in performance.

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 collections of sorted runs with exponentially increasing sizes [52]. 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 [68]. Maintaining a separate Bloom filter [13] per sorted run optimizes point query throughput by avoiding unnecessary accesses to runs that do not contain the desired data. Range query performance, while not benefiting from the presence of traditional Bloom filters, depends on the LSM tree structure.

Tuning LSM trees. Increasing number of applications relying on LSM-based storage backends, has brought a lot of attention to the problem of of tuning LSM trees for optimal performance. The common characteristic of all these methods, is the *assumption that one has full knowledge of the expected workload and of the underlying system resources* (processing units, memory, storage). Given such knowledge, there are a lot of works that optimize memory allocation to Bloom filters across different levels, memory distribution between buffers and Bloom filters and merging policies (i.e., *leveling* or *tiering*) [25, 26]. Further LSM optimization efforts have lead to hybrid merging policies to allow for more fine-grained tunings [27, 28, 42], optimized memory allocation strategies [14, 48, 50], variations of Bloom filters [54, 84, 85], new compaction routines [5, 51, 67, 86], and exploitation of data characteristics [1, 64, 83].

The Only Certainty is Uncertainty. Even when accurate information about the workload and underlying hardware is available, tuning data systems in general is a notoriously difficult research problem [18, 21, 74]. The explosive growth in the public and private use of the cloud infrastructure for data management [37, 47, 65] exacerbates the problem because it increases the uncertainty and the variability of the workloads [22, 33, 39, 40, 56, 59, 63, 69–71, 82].

An Example. Before we describe our framework, we give an example that demonstrates how variation in the observed workloads relative to the expected workload – used to tune an LSM tree-based storage system – leads to suboptimal performance. Figure 1 shows a sequence of workloads executed over an LSM-based engine. The x-axis is denotes a sequence of workloads and the y-axis shows the average disk accesses per workload. The experiment is split in three sessions, where the first and the last sessions receive the expected workloads, while the second session receives a different workload. Although it has the same read-to-write ratio of queries 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 average I/Os increase dramatically in the second session although 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 change tunings during execution as it would require redistributing the allocated memory between different components of the tree and potentially changing the shape of the tree. Rather, 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 the suboptimal system performance due to the variability in the observed workloads, we depart from the classical view of database tunings which assumes that the expected workload is known. Rather, we incorporate uncertainty into our workload and we introduce *Endure* a new *robust tuning paradigm* and we apply it to LSM trees. We propose a formulation that asks for the LSM tree configuration that maximizes the worst-case throughput over all the workloads in the *neighborhood* of an expected workload. We call this problem the Robust Tuning problem. We use the notion of KL-divergence between probability distributions to define the size of this neighborhood, implicitly assuming that the uncertain workloads would likely be contained in the neighborhood. As the KL-divergence boundary condition approaches zero, our problem becomes equivalent to the classical optimization problem (termed, the Nominal LSMTuning 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 uncertainty region considered, the larger workload discrepancy robust tuning can absorb. Leveraging work on robust optimization from the Operations Research community [10–12], we are able to efficiently solve the Robust Tuning problem and find the robust tuning for LSM tree-based storage systems.

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 into LSM tuning and motivated by problem definitions in the operations-research community we formulate the Robust Tuning problem.
- Our problem formulation can be meta-tuned to a specific degree of uncertainty in the expected workload.
- Leveraging results from the operations-research community, we show that we can find a robust LSM tuning efficiently. Our algorithm is also easy to implement in practice.
- We augment the existing analytical cost models of LSM tree-based storage systems with more precise methods to compute costs of workload execution.
- We show that robust tuning matches classical tuning when there is no uncertainty, while providing up to 95% higher throughput on average when faced with uncertain workloads.
- We integrate the proposed framework in RocksDB, a state-of-the-art LSM-based engine, and we validate the performance benefits of our robust tunings.
- We make our robust tuning framework publicly available to encourage reproducible research [6].

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 buffered in a memory buffer, and once full, its contents are sorted based on the key, forming an *immutable sorted run.* This run is then flushed to the first level on secondary storage. Each level of sorted runs has a maximum permitted size, which is tunable. Overall, for an LSM tree with L disk-resident levels, we denote the memory buffer as Level 0, and the remaining levels in storage, level 1 to L. The disk-resident sorted runs have exponentially increasing sizes following a tunable size ratio T. Figure 2 shows an overview of an LSM tree.

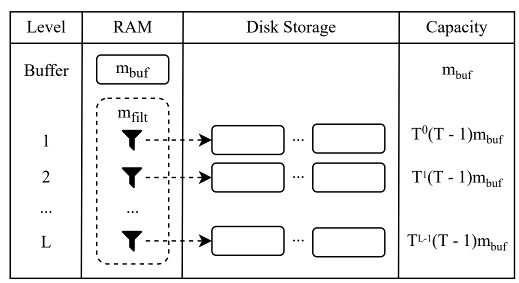


Figure 2: Overview of the structure of an LSM tree

We denote the number of bits of main memory allocated to the buffer as m_{buf}, which holds a number of entries with fixed entry size E. For example, in RocksDB the default buffer size is $m_{\mathrm{buf}} = 64$MB, and depending on the application, the entry size typically varies between 64B and 1KB.

Level 0 can be updated in-place since it is in memory, however, the runs in levels 1 and beyond are immutable. Each Level i has a capacity threshold of $(T-1)T^{i-1} \cdot \frac{m_{\mathrm{buf}}}{E}$ entries , thus, levels have exponentially increasing capacities by a factor of T. The total number of levels L for a given T is

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

where N is the total number of physical entries across all levels including updates and deletions [26, 54, 67].

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, 58]. 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, or (c) range queries.

Writes: A write operation is handled by a buffer append, and if the buffer gets full, it triggers a compaction as discussed above. Any write may include either a new key-value pair, a key-value pair that invalidates an existing one (an *update*), or a special key-value pair that deletes an existing one (a *delete*).

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 level to the largest one. For tiering, within a level, a lookup moves from the most to the least recent tier. The lookup terminates when it finds the first matching entry. Note that a point query might return an *empty* result or a *non-empty* result. We differentiate the two because, in general, workloads with empty point queries can be further optimized [25, 26].

Range Queries: A range lookup returns the most recent versions of the target keys by sort-merging the qualifying key ranges across all runs in 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 [25, 32]. 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 [25], 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 environmental setup is 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 [25, 48, 50], while often the size ratio and the merging strategy was also co-tuned [26]. In addition, recent work has introduced new hybrid merging strategies [27, 28, 68], and optimizations for faster data ingestion [53] and performance stability [51].

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* and the *system parameters*. One can think of the design parameters as those that someone who aims to optimize the performance of an LSM tree can tune, while the system parameters are given and therefore untunable.

Design Parameters. The design parameters we consider in this paper (in accordance with the related work [25, 26, 52]) 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 policy (π) as shown in Table 1. Recall that the policy refers to either leveling or tiering, as discussed in the previous section.

Term	Definition
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*)

Table 1: *Design* parameters of an LSM tree.

System Parameters. A complicated data structure like LSM trees also has other various *system parameters* and other non-tunable as shown in Table 2 (e.g., total memory (m), size of data entries E, page size B, data size N).

Term	Definition
m	Total memory (Bloom filters+write 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

Table 2: *System* and untunable parameters of an LSM tree.

LSM Tree Configuration. In terms of notation we use Φ to denote the LSM tree tuning configuration which essentially 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 the 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, as shown in Table 3. A workload can therefore 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 easiness 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

Term	Definition
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 updates

Table 3: Parameters describing the *workload*.

configuration (Φ) and a workload ($\mathbf{w}$), the expected cost for the workload can be computed as:

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

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 LSMTuning problem, defined as follows:

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

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

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 [25, 26, 50].

3.3 The Robust Tuning Problem

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

In the Nominal LSMTuning problem, the designers assume perfect information about the workload for which to tune the system. For example, they may assume that the input vector $\mathbf{w}$ represents the workload for which they have to 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; which 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 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*

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

Note that the above problem definition intuitively states the following: it recognizes that the input workload $\mathbf{w}$ won't 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 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 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., $\mathbf{w}^\top \mathbf{e} = 1$. Thus, in order to define the uncertainty region $\mathcal{U}_\mathbf{w}$, we use the Kullback-Leibler (KL) divergence function [49]. KL-divergence for two probability distributions defined as follows:

Definition 1. *The KL-divergence distance between two vectors* $\vec{p} = (p_1, \cdots, p_m)^\top \geq 0$ *and* $\vec{q} = (q_1, \cdots, q_m)^\top \geq 0$ *in* $\mathbb{R}^m$ *is defined as,*

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

Note that we could have potentially used other divergence functions [61] instead of the KL-divergence. We use the KL-divergence as we believe it fits our goal and intuitive understanding of the space of workloads.

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

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

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, that intuitively defines the neighborhood around the expected workload. This ρ can be computed as the mean KL-divergence from the historical workloads.

In terms of notation, ρ input 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:

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

Note that the above equation is a simple rewrite of Equation (6) that captures the intuition that the optimization is done over the *worst-case* workload among all the workloads in the uncertainty region $\mathcal{U}_{\mathbf{w}}$. An equivalent way of writing Equation (6) is by introducing an additional variable $\beta \in \mathbb{R}$, then writing the following:

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

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,

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

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

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

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 [9],

Lemma 4.1 ([9]). *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.* [9], 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, as 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.* [9] 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 with respect to 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 [79]. 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.

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, 54], 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 a small fraction of false positives. In particular, a point lookup probes a run's filter before accessing the run in secondary storage. If the filter returns negative, the target key does not exist in the run, and so the lookup skips accessing the run and saves one I/O. If a filter returns positive, then the target key may exist in the run, so the lookup probes the run at a cost of one I/O. If the run actually contains the key, the lookup terminates. Otherwise, we have a *false positive* and the lookup continues to probe the next run. False positives increase the I/O cost of lookups. The false positive rate (ϵ) of a standard Bloom filter desinged to query $\mathbf{n}$ entries using a bit-array of size $\mathbf{m}$ is shown by [77] 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 [80].

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 [25, 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

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

Additionally, false positive rates for all levels satisfy $0 \le f_i(T) \le 1$. 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 Tables 1, 2, and 3 of 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 detailed 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 to visit all levels (and every sorted run of every level for tiering) where false positives in the Bloom filters trigger I/O operations. Thus, the expected number of I/O operations per level depend 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:

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

In the leveling policy, each level has exactly one run. On the other hand, with tiering policy, each level has up to $(T-1)$ runs. All runs at the same level in tiering 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 $\frac{(T-1)T^{i-1}}{N_f(T)} \frac{m_{\text{buf}}}{E}$, where $N_f(T)$ denotes the number of entries in a tree completely full upto $L(T)$ levels. Thus,

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

Second, we assume that all levels preceding level i trigger I/O operations with probability equivalent to the false positive rates of the Bloom filters at those levels. Similar to the 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/O operations. 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:

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

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:

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

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 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):

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

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[1]. This is the exact computation of the cost done in Equation (2) and in the definitions of the Nominal LSMTuning and Robust Tuning problems (Equations (3) and (4) respectively).

6 UNCERTAINTY BENCHMARK

In this section, we describe an 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.

[1] Note that $z_0 + z_1 + q + w = 1$

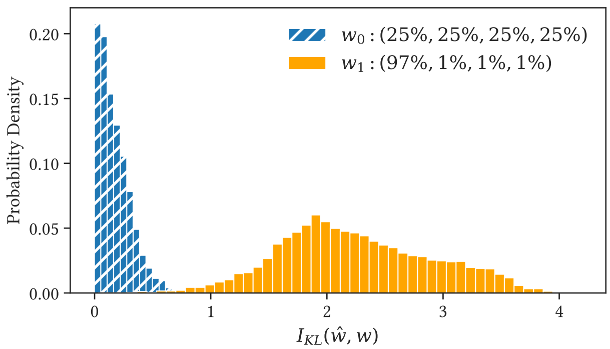


Figure 3: KL-divergence $I_{KL}(\hat{\mathbf{w}}, w)$ histograms of the sampled workloads wrt. to expected workloads $\mathbf{w}_0$ and $\mathbf{w}_1$.

Expected Workloads. We create robust tunings configurations for 15 expected workloads encompassing different proportions of query types. We catalog them into *uniform*, *unimodal*, *bimodal*, and *trimodal* categories based upon the dominant query types. A minimum 1% of each query type is always included in every expected workload to ensure a finite KL-divergence. Henceforth, we use $\mathbf{w}_i$ to refer to the i-th expected workload from Table 4.

Index	(z_0, z_1, q, w)				Type
0	25%	25%	25%	25%	**Uniform**
1	97%	1%	1%	1%	**Unimodal**
2	1%	97%	1%	1%	
3	1%	1%	97%	1%	
4	1%	1%	1%	97%	
5	49%	49%	1%	1%	**Bimodal**
6	49%	1%	49%	1%	
7	49%	1%	1%	49%	
8	1%	49%	49%	1%	
9	1%	49%	1%	49%	
10	1%	1%	49%	49%	
11	33%	33%	33%	1%	**Trimodal**
12	33%	33%	1%	33%	
13	33%	1%	33%	33%	
14	1%	33%	33%	33%	

Table 4: Expected workloads.

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.

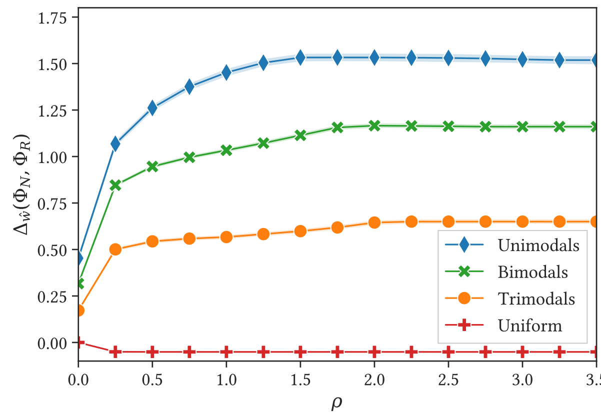


Figure 4: Delta throughput $\Delta_{\hat{\mathbf{w}}}(\Phi_N, \Phi_R)$ between the nominal and expected tunings across various expected workloads.

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. As an example, in Figure 3, we plot the distribution of KL-divergences of sampled workloads in $\mathcal{B}$ wrt. the expected workloads $\mathbf{w}_0$ and $\mathbf{w}_1$ from Table 4.

7 MODEL EVALUATION

Using the uncertainty benchmark defined in Section 6, and the analytical cost model from Section 5, we now rigorously evaluate the performance of the proposed robust tunings by Endure.

7.1 Evaluation Metrics

Now, we describe the metrics used to compare different tuning configurations.

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 wrt. 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 wrt. 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.

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

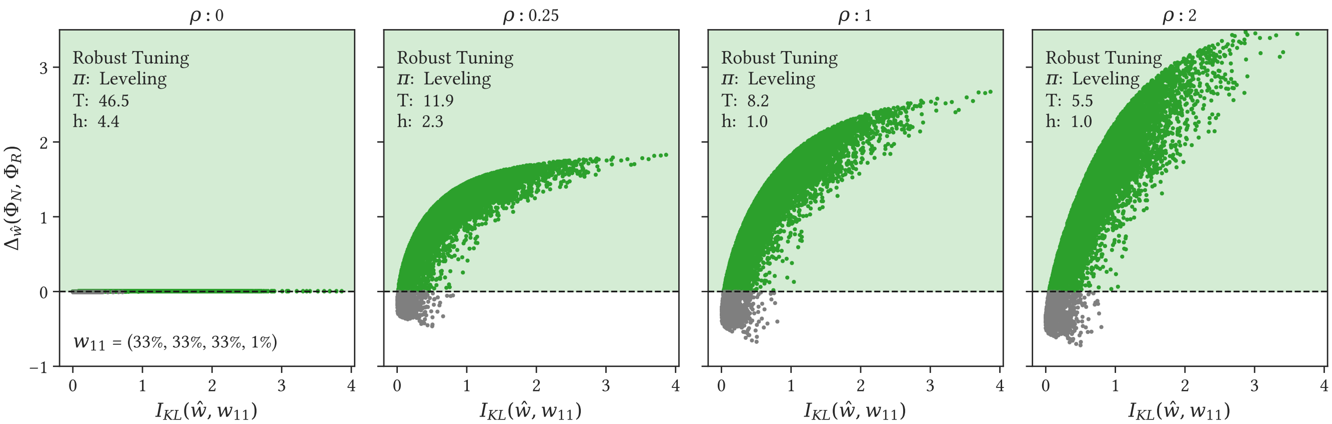


Figure 5: Impact of ρ on normalized delta throughput $\Delta_{\hat{\mathbf{w}}}(\Phi_N, \Phi_R)$ for tunings with expected workload $\mathbf{w}_{11}$.

model. For each of the expected workloads in Table 4, we obtain a single nominal tuning configuration (Φ_N) by solving the Nominal LSMTuning 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. Analysis presented in the following sections assumes a total available memory of 10 GB. In the following sections, for brevity purposes, we present representative results corresponding to individual expected workloads and specific system parameters. However, we exhaustively confirmed that changing these parameters do 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 [79], we obtain both nominal and robust tunings with the runtime for the above experiment being less than 10 minutes.

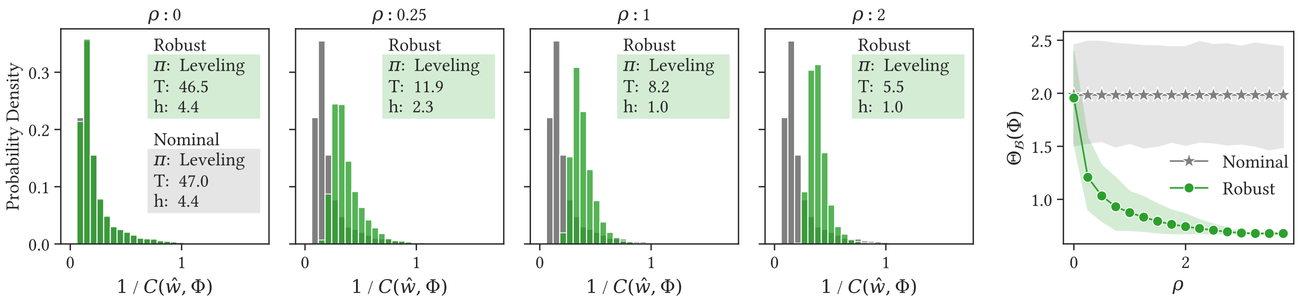
Comparison of Tunings. First, we address the question – *is it beneficial to adopt robust tunings relative to the nominal tunings?* Intuitively, it should be clear that the performance of nominally tuned databases would degrade when the workloads being executed on the database are significantly different from the expected workloads used for tuning. In Figure 4, 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 case of the uniform expected workload $\mathbf{w}_0$, Figure 3 shows us that instances of high values of KL-divergence are extremely rare. 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 5, 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 ρ = 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 wrt. expected workload increases. Furthermore, the robustness of such configurations have logically sound explanations. The expected workload in Figure 5 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 increasing percentage of write queries and hence limit the size-ratio to achieve overall higher throughput.

In Figure 6, we show the impact of tuning parameter ρ on the throughput range. Specifically, in Figure 6a 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 6b, 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 throughout over all $\hat{\mathbf{w}} \in \mathcal{B}$, but also a more consistent performance with lower variance when compared to the nominal tunings.

Choice of Tuning Parameter ρ. Now, we address the question – *What is the appropriate choice for the value of uncertainty parameter*



(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 6: Impact of ρ on throughput.

ρ? We provide guidance on the choice of ρ in absence of perfect knowledge regarding the future workloads that are likely to be executed on the database. Intuitively, we expect the robust tunings to be only weak when they are tuned for either too little or too much uncertainty. In Figure 7, 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 viz., (1) when 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 in workloads is higher. Based on this evidence, we can advise a potential database administrator that mean KL-divergences between pairs of historically observed workloads would be a reasonable value of ρ while deploying robust tunings in practice.

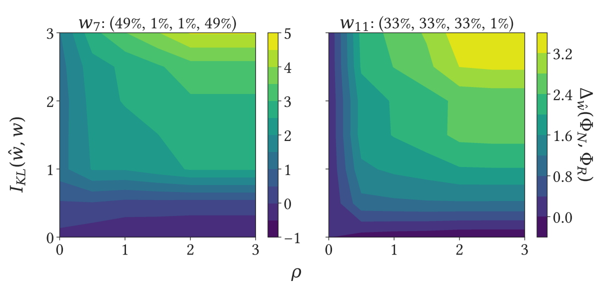


Figure 7: Delta throughputs $\Delta_{\hat{\mathbf{w}}}(\Phi_N, \Phi_R)$ for ρ vs $I_{KL}(\hat{\mathbf{w}}, \mathbf{w})$.

8 SYSTEM EVALUATION

Having studied the robust tunings analytically, we are now in a position to test our approach with a deployment in a full-blown LSM-based storage engine. In this section, we provide the details regarding the integration of Endure tunings to RocksDB and the performance benchmarking we followed to collect empirical results.

8.1 Experimental Setup & Implementation

Our server is powered by two Intel Xeon Gold 6230 processors and has 384 GB of main memory alongside a 1 TB Dell P4510 NVMe drive. It runs CentOS 7.9.2009 with a default page size of 4 KB. We use Facebook's RocksDB database, a popular LSM tree-based storage system, to evaluate our approach [31]. We use RocksDB's event hooks to implement both classical leveling and tiering policies despite its limited support for pure tiering compaction policy [66]. Following the Monkey memory allocation scheme [25], we allocate different bits per element for Bloom filters per level using the built-in implementation of RocksDB.

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 while 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 entries 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 $\mathcal{B}$. 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 actual 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, essentially reading zero or one page per level. While generating actual short-range queries, we randomly select a sequence of size B (number of entries that fit in a page) from the existing keys. Lastly, write queries consist of randomly generated key-value pairs, and thus they may update existing keys already present in the database.

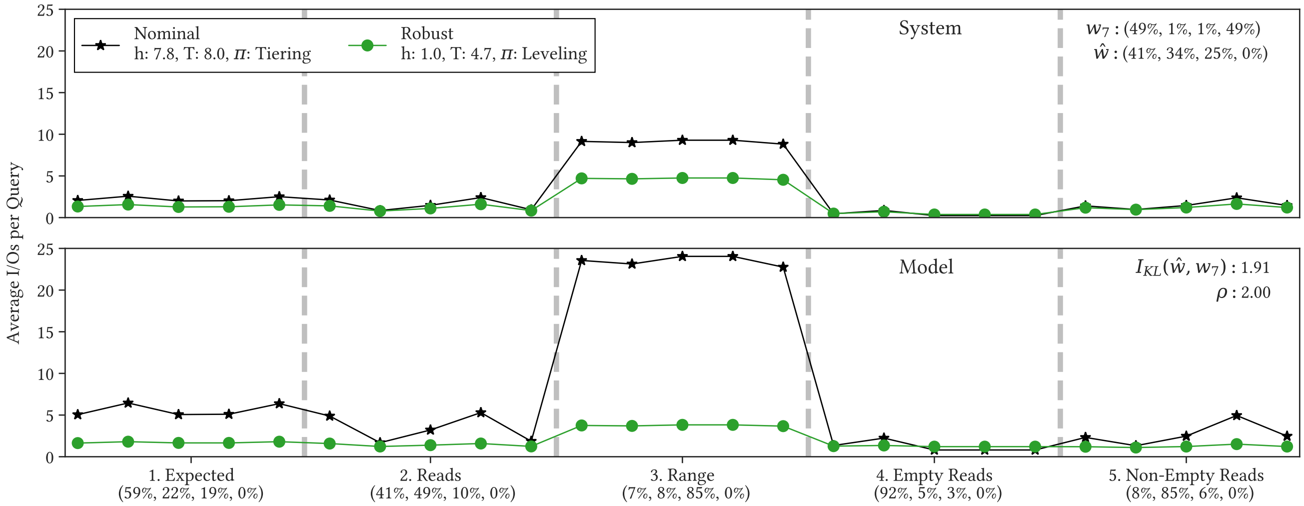


Figure 8: System (top) and model (bottom) performance for robust and nominal tunings in a read-only query sequence. Here the tuning parameter ρ closely matches the observed value of $I_{KL}(\hat{w}, w_7)$. Each session contains the label and average workload.

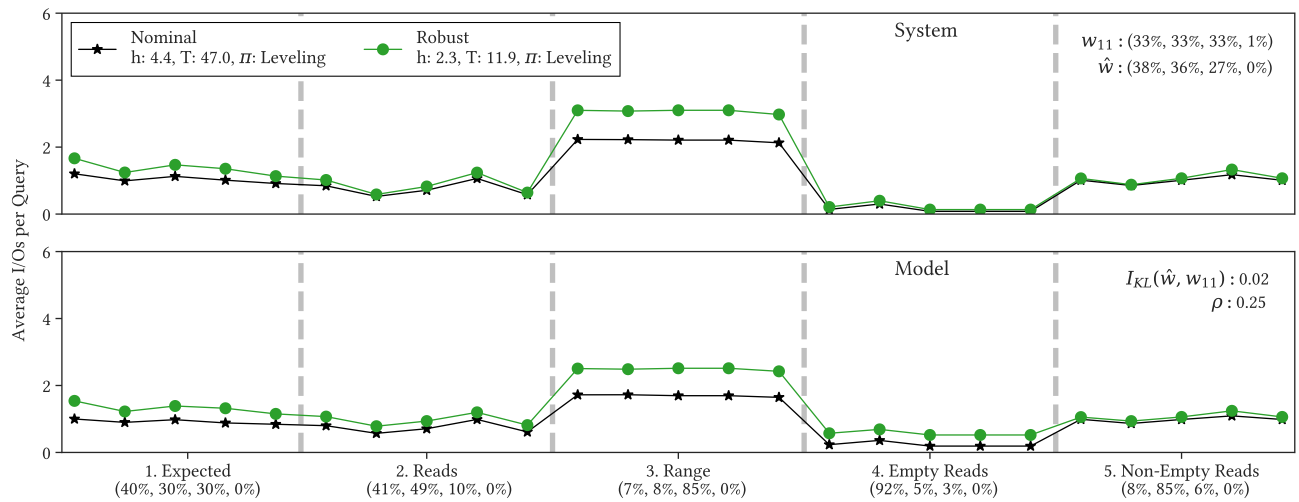


Figure 9: Read-only query sequence when the observed workloads $\hat{w}$ do not deviate from the expected.

8.3 Empirical Measurements

We now provide details about the precise empirical measurements during the execution of workloads on RocksDB. To obtain an accurate count of block accesses in our experiments, we turn off the block cache of RocksDB, and enable direct I/Os for all reads. We use the internal RocksDB statistics module to measure number of logical block accesses during reads, bytes written or flushed during writes, and bytes read and written during compactions. The number of logical blocks accessed during writes is then calculated by dividing the number of bytes reported by the default page size. It should be noted that precise measurement of write cost is difficult as writes inherently change the structure of the database, thereby affecting costs associated with all future queries. To estimate the amortized cost of writes, we compute 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.

8.4 Results

We first validate the analytical model discussed in Section 5 and then replicate key insights from Section 7 on a RocksDB database. In our implementation, the instantiation of RocksDB database combined with bulk loading requires on an average 10 minutes, while individual workloads are executed in about 3 minutes on an average.

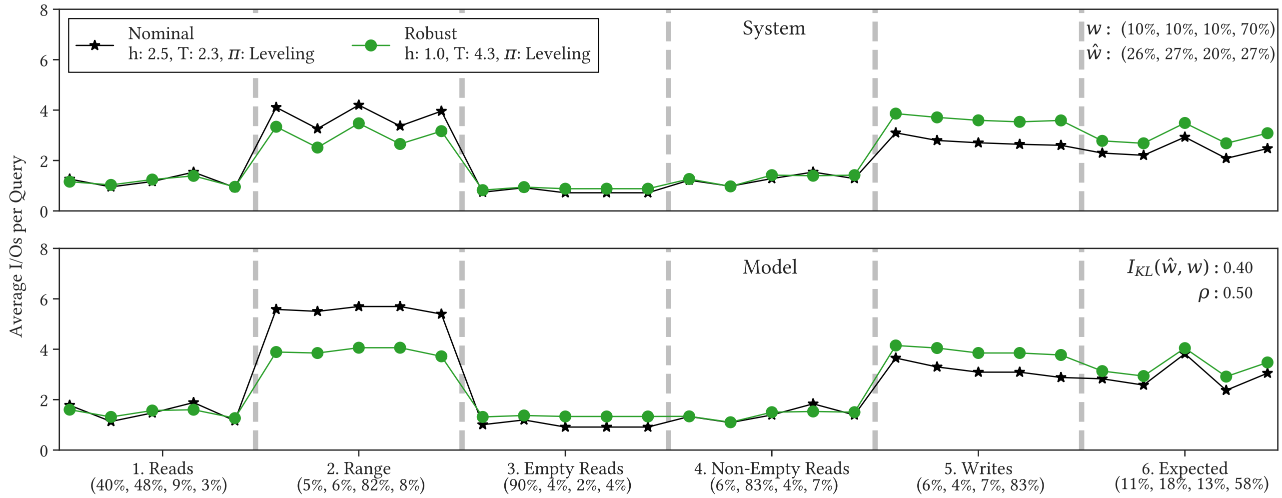


Figure 10: Read and write workload sequence where ρ and $I_{KL}(\hat{\mathbf{w}}, \mathbf{w})$ closely match.

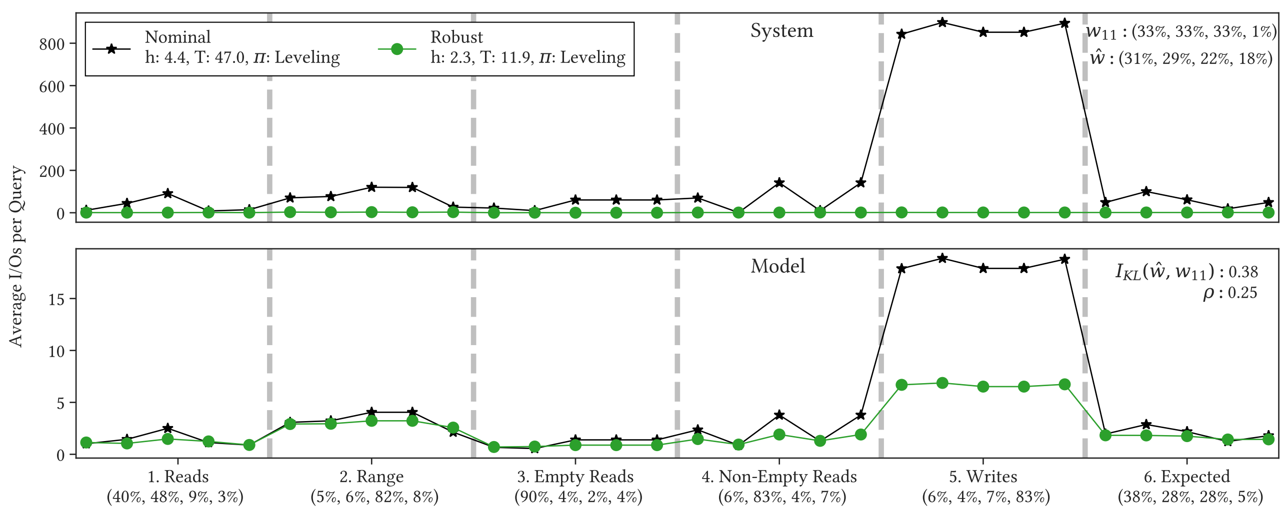


Figure 11: Read and write workload sequence where ρ and $I_{KL}(\hat{\mathbf{w}}, \mathbf{w})$ closely match.

Read Performance Validation. As discussed earlier in Section 8.3, write queries change the structure of the LSM tree and affect the costs of all following queries. Hence, we begin by first validating the accuracy of the analytical model for read queries in Figures 8 and 9 which exclude write sessions. In both figures, we compare actual I/Os per query on the database system (top) with the I/Os per query predicted by the model (bottom) for both nominal and robust tunings across different read session. We observe that the empirical measurements not only confirm the cost model predictions but also provide evidence in support of relative performance between different tunings. The discrepancy observed between the relative performances between the nominal and the robust tunings in the presence of range queries is due to the fence pointers in RocksDB. The analytical model does not account for the fence pointers which reduce the measured I/Os for (some) short range queries, thereby increasing the predicted I/Os.

Write Performance Validation. Now, we validate the write-portion of the analytical model by introducing an additional write query dominated session in Figures 10 and 11. Note that the structure of the LSM tree is continually changing across all the session in these figures due to the compactions caused by the write queries. For example, the dips in measured I/Os in the range query session in Figure 10 are the result of compactions triggered by write queries in preceding workloads leading to empty levels. Moreover, in Figure 11, the large size-ratio T leads to a shallow tree with extremely large levels. Thus, a compaction occurring in the write query dominated session triggers a sort at the lower levels of the tree resulting in a higher number of I/Os than predicted by the model. Overall, Figures 8– 10 confirm that our analytical model can accurately capture the relative performance of different tunings.

Choice of Tuning Parameter ρ Validation. In the model evaluation (Figure 7 in Section 7.3), 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 validation experiments described above. Specifically, Figures 8, 10, and 11 show instances where the KL-divergence of the observed workload averaged across all the sessions w.r.t. the expected workload is close to the tuning parameter ρ. In each of these experiments, the robust tuning outperforms the nominal. Conversely, in Figure 9, we find that the observed workloads are very similar to the expected workload ($I_{KL}(\hat{\mathbf{w}}, \mathbf{w}_{11}) = 0.2$), resulting in a performance benefit in favor of the nominal tuning, as predicted in Figure 7.

8.5 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 a steady-state performance. In Section 8.4, 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 of these 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 section, are easily explained based on the structure of the LSM tree.

One of the key takeaways of applying robust tuning to LSM trees is that *the leveling policies are inherently more robust* to perturbations in workloads, when compared to pure tiering policies. This observation is in line with the industry practice of deploying leveling or hybrid policies over pure tiering policies. Overall, based on our analytical and empirical results, robust tuning should always be employed when tuning an LSM tree to obtain robust tuning configurations, 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 which 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, 20]. 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, 19, 23, 78, 87], and periodic online workload analysis [16, 69–71] to capture workload drift [40]. In addition, there has been research on reducing the magnitude of the search space of tuning [15, 24] and on deciding the optional data partitioning [8, 60, 73, 75, 76]. 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 [35, 36, 43, 72]. 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, 55, 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 [57] is the first attempt to use unconstrained robust optimization to find a robust physical design. Cliffguard targets vertical partitioning decisions and is limited to unconstrained cases without setting service-level agreements. Perhaps the biggest difference between our approaches lies in the definition of the uncertainty region around a workload. Cliffguard algorithm assumes that the workloads are present in a Euclidean distance space, while our approach uses a more intuitive distance measure for variations in proportions of different queries in a workload viz., KL-divergence. In contrast to Cliffguard, our formulation of the robust problem does not require additional hyperparameters, such as the gradient descent step-size.

10 CONCLUSION

In this work, we explore the impact of workload uncertainty on the performance of LSM tree-based databases, and introduce Endure, a robust tuning paradigm that recommends optimal designs to mitigate any performance degradation in the presence of workload uncertainty. We show that in the presence of uncertainty, robust tunings increase database throughput compared to standard tunings by up to 5×. Additionally, we provide evidence that our analytical model closely matches the behavior measured on a database system. Endure can be used as an indispensable tool for database administrators to both evaluate the performance of tunings, and recommend optimal tunings in a few seconds without resorting to expensive database experiments.

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] Anonymous Authors. 2021. Robust LSM Tuning. https://anonymous.4open.science/r/robust-lsm-tuning-E212

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

[8] 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.

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

[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] 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

[18] Surajit Chaudhuri, Beno\^\it 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

[19] 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

[20] 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

[21] 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

[22] 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)*.

[23] 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

[24] 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

[25] 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

[26] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Transactions on Database Systems (TODS)* 43, 4 (2018), 16:1–16:48. https://doi.org/10.1145/3276980

[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. [n. d.]. MyRocks. *http://myrocks.io/* ([n. d.]).

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

[33] 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

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

[35] 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

[36] 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.

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

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

[39] 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.

[40] 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

[41] 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

[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 (SIGMOD)*.

[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] Intel. 2011. Best Practices for Building an Enterprise Private Cloud. *White Paper* (2011).
[48] 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
[49] S. Kullback and R. A. Leibler. 1951. On Information and Sufficiency. *The Annals of Mathematical Statistics* 22, 1 (1951), 79 – 86. https://doi.org/10.1214/aoms/1177729694
[50] 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
[51] 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.
[52] 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
[53] 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
[54] 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
[55] 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
[56] 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)*.
[57] 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
[58] 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
[59] 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
[60] 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
[61] Leandro Pardo. 2018. *Statistical Inference Based on Divergence Measures*. Chapman and Hall/CRC. https://doi.org/10.1201/9781420034813
[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] RocksDB. 2020. Leveled Compaction. *https://github.com/facebook/rocksdb/wiki/Leveled-Compaction* (2020).
[67] 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
[68] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *Proceedings of the VLDB Endowment* (2021).
[69] 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
[70] 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
[71] 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
[72] 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
[73] 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
[74] 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)*.
[75] 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
[76] 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
[77] 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
[78] 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
[79] 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
[80] Wikipedia contributors. 2021. Bloom filter — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Bloom_filter&oldid=1025193696. [Online; accessed 8-June-2021].
[81] WiredTiger. 2021. Source Code. *https://github.com/wiredtiger/wiredtiger* (2021).
[82] 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
[83] 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.
[84] 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
[85] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Succinct Range Filters. *ACM Transactions on Database Systems (TODS)* 45, 2 (2020), 5:1—-5:31. https://doi.org/10.1145/3375660
[86] 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 *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*. 225–237.
[87] 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