The VLDB Journal
https://doi.org/10.1007/s00778-023-00826-9

SPECIAL ISSUE PAPER

# Towards flexibility and robustness of LSM trees

**Andy Huynh**[1] · **Harshal A. Chaudhari**[1] · **Evimaria Terzi**[1] · **Manos Athanassoulis**[1]

Received: 31 January 2023 / Revised: 19 August 2023 / Accepted: 29 October 2023


**Abstract**
Log-structured merge trees (LSM trees) are increasingly used as part of the storage engine behind several data systems, and are frequently deployed in the cloud. As the number of applications relying on LSM-based storage backends increases, the problem of performance tuning of LSM trees receives increasing attention. We consider both *nominal* tunings—where workload and execution environment are accurately known a priori—and *robust* tunings—which consider *uncertainty* in the workload knowledge. This type of workload uncertainty is common in modern applications, notably in shared infrastructure environments like the public cloud. 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 policy, size ratio, and memory allocation on the overall performance. ENDURE considers a robust formulation of the throughput maximization problem and recommends a tuning that offers near-optimal throughput when the executed workload is not the same, instead in a *neighborhood* of the expected workload. Additionally, we explore the robustness of flexible LSM designs by proposing a new unified design called K-LSM that encompasses existing designs. We deploy our robust tuning system, ENDURE, on a state-of-the-art key-value store, RocksDB, and demonstrate throughput improvements of up to 5× in the presence of uncertainty. Our results indicate that the tunings obtained by ENDURE are more robust than tunings obtained under our expanded LSM design space. This indicates that robustness may not be inherent to a design, instead, it is an outcome of a tuning process that explicitly accounts for uncertainty.

**Keywords** Database · Database tuning · Robust tuning · LSM tree · Instance-optimized systems

## 1 Introduction

**Ubiquitous LSM-based key-value stores**. Log-Structured Merge trees (LSM trees) are commonly deployed as the back-end storage engine of modern key-value stores [61]. The high ingestion rates and fast reads provided by LSM trees have led to their wide adoption by systems like RocksDB [32] at Meta, LevelDB [34] and BigTable [18] at Google, HBase [7], Cassandra [8] at Apache, WiredTiger [86] at MongoDB, X-Engine [40] at Alibaba, and DynamoDB [30] at Amazon.

✉ Andy Huynh
ndhuynh@bu.edu

Harshal A. Chaudhari
harshal@bu.edu

Evimaria Terzi
evimaria@bu.edu

Manos Athanassoulis
mathan@bu.edu

[1] Boston University, Boston, USA

LSM trees store incoming data within a memory buffer, which is subsequently flushed to storage when full, and merged with earlier buffers to form a collection of sorted runs with exponentially increasing sizes [56]. Frequent merging of sorted runs leads to a higher merging cost, but facilitates faster lookups (*leveling*). On the flip side, lazy merging policies (*tiering*) trade lookup performance for lower merging costs [73].

**Tuning LSM trees**. As the number of applications relying on LSM-based storage backends increases, the problem of performance tuning LSM trees has garnered a lot of attention. A common assumption of these methods is that when creating an instance-optimized system [51], one has *complete knowledge of the expected workload and the execution environment*. Given such knowledge, prior work focuses on optimizing LSM tree parameters such as memory allocation for 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

Published online: 11 January 2024

with more fine-grained tunings [28, 29, 42]; optimized memory allocation strategies [14, 49, 53], Bloom filter variations [58, 89], new compaction strategies [5, 54, 72, 73, 90], and exploitation of data characteristics [1, 67, 88].

Even when accurate information about the workload and underlying hardware are available, tuning data systems is a notoriously difficult research problem [19, 22, 79]. Additionally, the explosive growth in the usage of the cloud infrastructure for data management [37, 68] has exacerbated this problem due to the increase in uncertainty and variability in workloads [23, 33, 38, 39, 62, 66, 74–76, 87].

To address this challenge, we introduce ENDURE,[1] a general framework for providing robust tunings under uncertain input workloads. ENDURE introduces a tuning-under-uncertainty paradigm by formulating the classic tuning problem as a robust optimization problem. Our experiments demonstrate the benefits of robust tunings compared to existing baselines for tuning LSM trees.

**Expanding the LSM design space**. In this paper, we build on our prior work [41] and expand it along multiple dimensions. We take a more critical look at the performance of LSM tunings for flexible LSM designs both with and without workload uncertainty. After careful consideration of existing LSM designs and tuning approaches—e.g., Monkey [26] and Dostoevsky [28]—we propose a general and more unified LSM design, termed K-LSM. Our design allows each level to have a variable number of potentially overlapping files. Therefore, we can describe both standard compaction policies (i.e., tiering [48] and leveling [61]), and existing hybrid compaction policies. We demonstrate that K-LSM can reduce to data layouts such as Lazy Leveling [28, 29], Dostoevsky [28], and 1-Leveling [73], thus making it a unified LSM design. Furthermore, we accompany the K-LSM design with a cost model, which in turn can capture the costs of all aforementioned approaches.

**Performance of LSM tunings**. Next, we check the feasibility of tuning an LSM tree under the assumption of an accurately known workload (no uncertainty) using the K-LSM cost model. We show that this can be done using off-the-shelf numerical solvers. Our experiments indicate that tunings obtained using flexible designs provide better system performance when compared to those obtained from state-of-the-art LSM designs. To the best of our knowledge, we are the first to propose a unified LSM cost model.

**Robustness of LSM trees**. In the second part of the work, we present results with ENDURE [41], our system for *robust* LSM tree tuning—i.e., LSM tree tuning in the presence of workload uncertainty. Here, we depart from the classical view of database tuning, which assumes accurate knowledge about the expected workload. Toward this, ENDURE introduces a new *robust tuning paradigm* that incorporates expected uncertainty into optimization and applies 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 and Machine Learning communities [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 [60]. However, this methodology is not well-suited for the LSM tuning problem. We provide additional details regarding this in Sect. 12.

**Flexibility in design and robustness**. Finally, we experimentally investigate whether the nominal tunings obtained by various LSM designs are inherently robust. That is, we investigate whether the lack of robustness in the state-of-the-art nominal tunings is a consequence of the designs not being expressive enough, or a result of the tuning process's lack of consideration for uncertainty. Our findings indicate that the nominal tunings obtained via K-LSM provide a benefit over traditional LSM designs in scenarios where the workload does not deviate from the expected. However, this benefit does not appear in the contrasting scenario where the workload does deviate from the expected. Rather, tunings obtained from ENDURE exhibit higher throughput with simpler LSM designs than nominal tunings with flexible LSM designs. Hence, we conclude flexibility does not inherently provide robustness.

**Contributions**. To the best of our knowledge, our work is the first that presents a unified LSM design with an associated cost model that is a generalization of all the existing state-of-the-art approaches. Moreover, we present the first systematic approach to selecting a robust tuning for instance-optimized LSM tree-based key-value stores under workload uncertainty, utilizing robust optimization techniques from machine learning. Finally, we are the first to explore whether the robustness of an LSM tree can be an inherent design property or a result of explicitly tuning for uncertainty.

[1] An earlier version of this work appeared in VLDB 2022 [41].

Our technical and practical contributions can be summarized as follows:

- We introduce K-LSM, a new unified LSM design that describes both classic designs and recently proposed state-of-the-art hybrid designs (§4). We present its implications on classical LSM tuning (§5).
- We incorporate workload uncertainty into LSM tunings and provide algorithms to compute robust tunings efficiently. ENDURE can be tuned for varying degrees of workload uncertainty, and is practical enough to be adopted by the current state-of-the-art LSM storage engines (§6).
- We develop an uncertainty benchmark that can evaluate the robustness of the current state-of-the-art LSM-based systems (§7).
- In our model-based analysis, we show that robust tunings obtained from ENDURE provide up to 5× higher throughput when faced with uncertain workloads (§8).
- We deploy and test ENDURE in RocksDB, a state-of-the-art LSM storage engine, to demonstrate the feasibility of using robust tunings on commercial systems. We show that ENDURE achieves up to 2.4× throughput speedups, and these gains are independent of the database size (§9).
- By evaluating the robustness of the K-LSM design, we demonstrate that *robustness is not inherent to a design,* rather is an outcome of a tuning process that accounts for uncertainty (§10).
- To encourage reproducible research, we make all our code publicly available.[2]

## 2 Background on LSM trees

**Basics LSM trees use the out-of-place**. paradigm to store key-value pairs [56]. Inserts, updates, and 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 subsequently organized into logical levels.

Thus, for an LSM tree with $L$ disk-resident levels, we label the memory buffer as Level 0 and the remaining levels in storage from 1 to $L$. The disk-resident levels have exponentially increasing sizes dictated by a tunable size ratio $T$. Figure 1 shows an overview of an LSM tree.

We denote the number of bits of main memory allocated to the buffer $m_{\text{buf}}$, which holds several entries with a fixed entry size $E$. For example, in RocksDB, the default buffer size is $m_{\text{buf}} = 64\text{MB}$, and depending on the application, the entry size typically varies between 64B and 1KB. The buffer at Level 0 is mutable and can be updated in place, while runs

[2] https://github.com/BU-DiSC/endure.

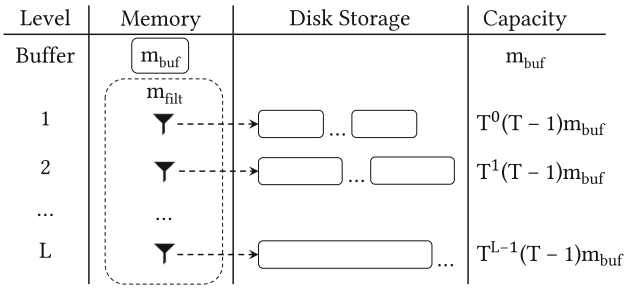



**Fig. 1** Overview of the structure of an LSM tree

starting at Level 1 and beyond are *immutable*. Each Level $i$ has a capacity threshold of $(T-1)\cdot T^{i-1}\cdot \frac{m_{\text{buf}}}{E}$ entries, thus, the 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 [26, 58, 72].

**Compaction policies: leveling and tiering**. Classically, LSM trees support two compaction policies: leveling and tiering [56, 70]. In leveling, each level contains at most one run, and every time a run in Level $i-1$ ($i \geq 1$) is flushed to Level $i$, it greedily sort-merges (compacts) with the run from Level $i$, assuming it exists. With tiering, every level must accumulate $T$ runs before a compaction is triggered. During a compaction, entries with a matching key are consolidated, and only the most recent valid entry is retained [31, 61, 73].

**Flexible compaction policies**. The different LSM tree compaction policies form a continuum between a read-optimized and write-optimized data layout, where leveling and tiering policies are the two extremes [70]. Hybrid compaction policies allow a smooth transition of the tree shape to strike a balance between the read and write throughput [28, 29]. Lazy Leveling assigns the upper levels of the LSM tree to a tiering policy and the last level to a leveling policy to improve the worst-case cost for writes while maintaining near-optimal read performance. This is motivated by the fact that the last level statistically contains most of the LSM tree's data.

This notion of assigning different compaction policies per level is further expanded by the Dostoevsky design and the Fluid LSM tree [28]. Rather than assigning each level a different compaction policy, the Fluid LSM tree uses two limits for the number of runs per level, one for the last level of the LSM tree, and a different one for all the upper levels. This allows the Fluid LSM design to express fine-grained hybrid compaction policies between leveling and tiering.

In this work, we further expand this approach by proposing K-LSM, a more expressive LSM compaction model that unifies all prior approaches by allowing each level to parameterize its capacity in terms of the number of files it can hold.

In Sect. 4, we discuss K-LSM in detail and demonstrate that it can explore a wider design space. We further discuss its implications on the robustness of its tunings.

**LSM tree operations**. An LSM tree supports three basic operations: (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 called a tombstone that *deletes* an existing key.

*Point Queries:* A point query searches for the value of a specific key. It begins by looking at the memory buffer and then traverses the tree from the smallest to the largest level. At each level, the lookup moves from the most recent sorted run to the oldest sorted run, terminating 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 as it has been shown workloads with empty point queries can be further optimized [26].

*Range Queries:* A range query lookup returns the most recent version of all keys within the desired range by potentially scanning every run at every level.

**Optimizing lookups**. LSM tree lookups are optimized using *filters* and indexes (also termed *fence pointers*) [71]. In the worst case, a lookup needs to probe every run, however, LSM engines use one filter per run [26, 32] to reduce this cost. While the filters are part of each run, they are aggressively cached in memory. One of the most common filter designs used in LSM trees is the Bloom filter [13]. A Bloom filter is a probabilistic membership test data structure that responds with a false positive rate $f$, which is a function of the ratio between the number of memory bits allocated $m_{\text{filt}}$ and the number of elements indexed. By probing the Bloom filter of a particular level, an LSM tree can skip accessing that run altogether when it does not contain the indexed key. In practice, for efficient storage, Bloom filters are maintained at the granularity of files [31]. Fence pointers hold the smallest key for each disk page of all sorted runs into main memory [26] to quickly identify which page(s) to read for a lookup. In this work, we assume that fence pointers are required and consume a fixed amount of memory in the system. Therefore, any operation that requires a single I/O will only require one logical page lookup by the operating system by following the corresponding fence pointer. We further assume that a single I/O operation corresponds to exactly one logical page access.

**Tuning LSM trees**. An LSM tree is a highly tunable data structure where the size ratio, compaction policy, exact shape of the tree, and memory allocation can all be tuned. Classical LSM tuning strategies start with an offline analysis and assume the workload information and the execution environment are accurately known a priori to deployment. In comparison, online tuning strategies change LSM tuning knobs in response to workloads, however, the design parameters that mainly drive the performance must be dictated before deployment [50, 56]. While LSM trees are also deployed as collections that can be co-tuned [55], here we focus on deploying and tuning single instances of LSM trees. Under that assumption, LSM tree tuning considers the optimal allocation of available main memory between Bloom filters and buffering [49, 53], the optimal choice of size ratio, and the data layout strategy [26–28]. Such design decisions are common across industry-standard LSM-based engines, such as Apache Cassandra [8], AsterixDB [6], RocksDB [32], and InfluxDB [47]. Lastly, recent work has introduced new hybrid merging strategies [29, 73], and optimizations for faster data ingestion [57] and performance stability [54].

## 3 Preliminaries

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 a part of the system the LSM tree is deployed on, 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 ($m_{\text{filt}}$), the memory allocated to the write buffer ($m_{\text{buf}}$), and the compaction policy ($\pi$). These are ubiquitous design parameters and have been extensively studied as having the largest impact on performance [26, 56]. Therefore, we focus on these parameters to define a problem that is not bound to any specific LSM engine. Recall that the compaction policy refers to either leveling or tiering in a classical design, or may contain other parameters used to describe hybrid designs as we discuss in Sect. 4.2.

**System parameters**. In production deployments, performance depends on various *system parameters* (e.g., total memory $m$, page size $B$), and other non-tunable data-dependent parameters (e.g., data entry size $E$, amount of data $N$). We assume these parameters are known a priori and fixed throughout the tuning process.

**LSM tree configuration**. We use $\Phi$ 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 $m_{\text{filt}}$ and not $m_{\text{buf}}$, because the latter can be derived using the total available memory ($m_{\text{buf}} = m - m_{\text{filt}}$).

**Workload**. The choice of the parameters in $\Phi$ 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 is 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, 56]. Additionally, complex workloads (i.e., SQL state-

**Table 1** Summary of problem notation

| Type | Term | Definition |
|---|---|---|
| Design | $m_{\text{filt}}$ | Memory allocated for Bloom filters |
| | $m_{\text{buf}}$ | Memory allocated for the write buffer |
| | $T$ | Size ratio between consecutive levels |
| | $\pi$ | 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 nonzero-result point lookups |
| | $q$ | Percentage of range queries |
| | $w$ | Percentage of writes |

ments) generate access patterns of 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 of the same dimension as $\mathbf{w}$.

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

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

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

## 4 The cost model of LSM trees

In this section, we provide a detailed cost model that accurately captures the behavior of a wide collection of LSM compaction strategies, including classical leveling and tiering, as well as hybrid approaches. Following prior work [26, 58], we focus on the four types of operations described earlier: point queries that return an empty result, point queries that have a match, range queries, and writes.

### 4.1 Model basics

When modeling the read cost of LSM trees, a key quantity to accurately capture is the amount of superfluous I/Os that take place. Although Bloom filters are used to minimize extra I/Os, 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 over the assigned fence pointer saving a single random 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. Then, 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 number of I/Os. The false positive rate ($\epsilon$) of a standard Bloom filter that is designed to hold information over $n$ entries using a bit-array of size $m$ is given by [82]:

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

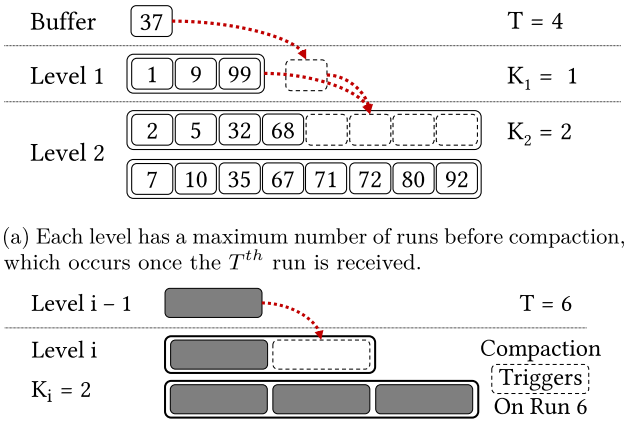
Note that the above equation assumes the use of an optimal number of hash functions in the Bloom filter [85].

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 $m_{\text{filt}}$ is the overall amount of main memory allocated to the filters. As $m_{\text{filt}}$ approaches 0 or infinity, the term $O\left(e^{-m_{\text{filt}}/N}\right)$ approaches 1 or 0 respectively. Here, we build on the state-of-the-art Bloom filter allocation strategy proposed in Monkey [26] that uses different false positive rates for each level 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

$$f_i(T) = \frac{T^{\frac{T}{T-1}}}{T^{L(T)+1-i}} \cdot e^{-\frac{m_{\text{filt}}}{N} \ln(2)^2}. \tag{3}$$

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 and system parameters**. In Sect. 3 we introduced the key design and system parameters needed to model LSM tree performance. In addition to those parameters, there are two more auxiliary and derived parameters we use: the potential storage asymmetry [64] in reads and writes ($f_a$) and the expected selectivity of range queries ($S_{\text{RQ}}$).




(a) Each level has a maximum number of runs before compaction, which occurs once the $T^{th}$ run is received.

**Fig. 2** K-LSM provides a flexible way to describe different compaction behaviors. In this figure, assume the buffer is the same size as a logical page; then each sorted run is composed of multiple pages

**Table 2** Additional model notation

| Term | Definition |
|---|---|
| $Z_0(\Phi)$ | Empty read cost w.r.t to a specific LSM configuration $\Phi$ |
| $Z_1(\Phi)$ | Non-empty read cost w.r.t to a specific LSM configuration $\Phi$ |
| $Q(\Phi)$ | Range read cost w.r.t to a specific LSM configuration $\Phi$ |
| $W(\Phi)$ | Write cost w.r.t to a specific LSM configuration $\Phi$ |
| $L(T)$ | Number of levels to fill a tree with size ratio $T$ |
| $N_f(T)$ | Number of entries to fill a tree with size ratio $T$ |
| $K_i$ | The maximum number of overlapping files for level $i$ |
| $f_i(T)$ | The Bloom filter false positive rate at Level $i$ with a size ratio $T$ |
| $f_a$ | Read/write Asymmetry ratio for storage device |
| $f_{\text{seq}}$ | Cost of a sequential read w.r.t a random read |
| $S_{\text{RQ}}$ | Range query selectivity |

### 4.2 Extending classic LSM compaction policies

We now introduce a new variable $K_i$ that denotes the maximum number of files for a given level $i$. It captures a unified design for both classical compaction policies (i.e., tiering and leveling) by introducing a range of new hybrid policies.

**Maximum files per level**. Figure 2 displays the basic structure of an LSM tree with $K_i$ assigned for all levels. We define $K_i$ as the maximum number of sorted immutable runs before a full-level compaction triggers, essentially the capacity of runs per level. In a classic tiering compaction policy, a single level of an LSM Tree traditionally has a max of $(T-1)$ runs, each of size $\frac{m_{\text{buf}}}{E} \cdot T^{(i-1)}$ where $i$ is the assigned level. A full-level compaction triggers once the level receives $T$ runs from the level above, as a result, a level will have at most $(T-1)$ runs. In our new design, each level still respects the maximum entry capacity for an LSM tree, as each run will have at most $\frac{m_{\text{buf}}}{E} \cdot \frac{T^{i-1}}{K_i}$ entries. Figure 2a shows an example of a tree right before the compaction occurs. Once the buffer is flushed, Level 1 will compact all runs within the level and send a sorted run to Level 2, which subsequently sort-merges the received run with the existing data. Then after four more buffer flushes, Level 2 will have received another run and trigger a compaction, creating a new Level 3.

**Compaction behavior**. When $K_i = T-1$ for level $i$, the design is equivalent to a tiering policy, while for $K_i = 1$, it is equivalent to a leveling policy. As incoming sorted runs are compacted, we choose not to split runs, rather, we only merge runs or logically move them from one level to another. Therefore, for values in between $T-1$ and 1, we alternate when compacted runs from the level above are merged or simply logically moved. For example, Fig. 2b shows a scenario for $K_i = 2$, and $T = 6$. The first three runs from the level above would be compacted to form a single run; the next 2 runs would merge to form a sorted run. In this instance, each run would be of a different size, one holding the equivalent of three compactions while the other holding 2. Otherwise, if $K_i$ and $T$ are set such that $(K_i - 1)/T$ is an integer, the size of each sorted run is equivalent. The sixth flush from Level $i-1$ triggers a full-level compaction flushing to Level $i+1$.

### 4.3 A general cost model

Using the above insights, we model the costs in terms of the expected number of I/O operations required for the fulfillment of the individual queries. We summarize new notations introduced for the cost model in Table 2.

**Expected empty point query cost**. $(Z_0)$ A point query that returns an empty result will have visited all sorted runs on every level and issue an I/O for every false positive result among the Bloom filters. Therefore, the expected number of I/Os per level depends on the Bloom filter memory allocation at that level. Hence, Eq. (4) expresses $Z_0$ in terms of the false positive rates at each level as:

$$Z_0(\Phi) = \sum_{i=1}^{L(T)} K_i \cdot f_i(T). \tag{4}$$

For each level, there will be at most $K_i$ runs, and each run will have equal false positive rates.

**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 full tree up to $L(T)$ levels:

$$N_f(T) = \sum_{i=1}^{L(T)} (T-1) \cdot T^{i-1} \cdot \frac{m_{\text{buf}}}{E}. \tag{5}$$

Second, we assume that all levels preceding Level $i$ will trigger an I/O operation 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 the preceding levels is $\sum_{j=1}^{i-1} f_j(T)$. Lastly, each level will contain at most $K_i$ sorted runs, we assume that on average the entry is found in the middle run resulting in an additional $\frac{(K_i-1)}{2} \cdot f_i(T)$ extra I/Os. Thus, we can compute the non-empty point query cost as an expectation over the entry being found at any of the $L(T)$ levels of the tree as follows:

$$Z_1(\Phi) = \sum_{i=1}^{L(T)} \frac{(T-1) \cdot T^{i-1}}{N_f(T)} \cdot \frac{m_{\text{buf}}}{E} \cdot \left(1 + \sum_{j=1}^{i-1} K_j \cdot f_j(T) + \frac{K_i - 1}{2} f_i(T)\right). \tag{6}$$

**Range queries cost**. ($Q$) A range query will issue at most one disk seek per run per level, or $K_i$ disk seeks. Each seek is then followed by a sequential scan. The cumulative number of pages scanned over all runs is $S_{\text{RQ}} \cdot \frac{N}{B}$, where $S_{\text{RQ}}$ is the average proportion of all entries included in range lookups. After finding the first valid page, range queries perform sequential I/Os for subsequent pages rather than a random I/O. Therefore, we add a scaling factor $f_{\text{seq}}$ that represents the cost of a sequential I/O with respect to one random I/O. Hence, the overall range lookup cost $Q$ in terms of logical pages reads is as follows:

$$Q(\Phi) = f_{\text{seq}} \cdot S_{\text{RQ}} \cdot \frac{N}{B} + \sum_{i=1}^{L(T)} K_i. \tag{7}$$

**Write cost**. ($W$) We model worst-case writing cost assuming that the vast majority of incoming entries do not overlap. This implies most entries will propagate through all levels of the LSM tree. Therefore, we calculate the expected number of I/Os by first estimating the average number of merge operations a single write participates in at Level $i$, and summing over all levels. We start by deriving the total number of merges that occur on Level $i$. note that Level $i$ will receive at most $T-1$ flushes from Level $i-1$ before a full level compaction is triggered. Additionally, a run at Level $i$ needs $\frac{T-1}{K_i}$ flushes from Level $i-1$ to reach its maximum size; we will refer to this as the flush capacity. Figure 3 shows the number of flushes and flush capacity for Level $i$.

Analyzing a single sorted run at Level $i$, we observe that the last flush will only participate in a single eager compaction as the sorted run will reach its flush capacity at that point. The second to last flush participates in 2 merges, the third to last in 3 merges, and the first flush in $\frac{T-1}{K_i} - 1$ merges as new flushes are eagerly compacted into the sorted run. Therefore, the total count of merge operations for $K_i$ sorted runs on Level $i$ is

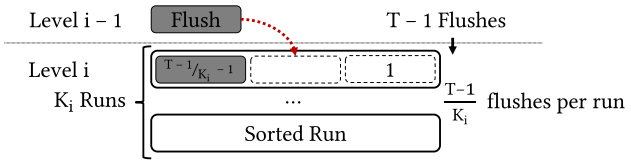



**Fig. 3** The last flush of a sorted run participates in 1 merge as it eagerly merges into the sorted run. The first flush will participate in all subsequent eager merges from new flushes

$$K_i \cdot \sum_{j=1}^{\frac{T-1}{K_i}-1} j = (T-1) \cdot \frac{(T-1-K_i)}{2K_i}. \tag{8}$$

Given the total merges for Level $i$, we can now calculate the average number of merges a single write participates in. First, we divide the total merges at Level $i$ from Eq. (8) by the number of flushes from Level $i-1$ ($T-1$) to receive an average merge count of $\frac{(T-1-K_i)}{2K_i}$. Second, to account for the final full-level merge that occurs on the $T^{th}$ flush, we add 1 additional merge. Therefore, the average number of merges, and subsequently I/Os, a single write participates in at Level $i$ is simply $\frac{T-1+K_i}{2K_i}$

To calculate the cost of a single insert, we need to divide the average number of merges every level by the number of entries per page, $B$. Additionally, as every compaction operation reads data at Level $i-1$ and writes to Level $i$, we model the potential asymmetry between reads and writes on the underlying storage device[3] using $f_a$. For example, a device for which a write operation is twice as expensive as a read operation has $f_a = 2$. When flushing the buffer, writes perform sequential I/Os as opposed to random I/Os, hence, we add $f_{\text{seq}}$ term to account for the cost of different I/O types. Summing the average I/Os per level for all levels, the total I/O cost is captured by:

$$W(\Phi) = f_{\text{seq}} \cdot \frac{1+f_a}{B} \cdot \sum_{i=1}^{L(T)} \frac{T-1+K_i}{2K_i}. \tag{9}$$

**Total expected cost**. The total expected operation cost, C(**w**, $\Phi$), is computed by weighing the empty point lookup cost $Z_0(\Phi)$ from Eq. (4), the non-empty point lookup cost $Z(\Phi)$ from Eq. (6), the range lookup cost $Q(\Phi)$ from Eq. (7), and the write cost $W(\Phi)$ from Eq. (9) by their proportion in

[3] Flash-based SSDs typically exhibit a read/write asymmetry, where writes are 2× to 10× more expensive than reads [64].

**Table 3** Variations of LSM data layouts

| LSM layout | Setting |
|---|---|
| Fluid LSM [28] | $K_1 = \ldots = K_{L-1}$ |
| Lazy Leveling [28, 29] | $K_L = 1, K_i = T - 1, \ \forall i \neq L$ |
| 1-Leveling [73] | $K_1 = T - 1, K_i = 1, \ \forall i \neq 1$ |
| Tiering [48] | $K_i = T - 1, \ \forall i$ |
| Leveling [61] | $K_i = 1, \ \forall i$ |
| K-LSM (§4) | All $K_i \in \mathbb{Z}$ |

the workload represented by the terms $z_0$, $z$, $q$ and $w$ respectively (note that $z_0 + z_1 + q + w = 1$).

### 4.4 Expressing LSM data layout variants

With the introduction of $K_i$, we can use our cost model to effectively describe the behavior of other common LSM designs. For example, for a classic leveling compaction policy we set $\forall i, K_i = 1$. This results in each level eagerly merging incoming compacted runs into a single run, which is equivalent to leveling. Additionally, our cost model can easily describe other flexible LSM compaction behavior. If we restrict all capacities before the last level to be equivalent (i.e., $K_1 = K_2 = \ldots = K_{L-1}$ where $L$ is the last level), our cost model expresses the Fluid LSM design as described in Dostoevsky [28]. With $K_L = 1$ and all $K_1 = K_2 = \ldots = K_{L-1} = T - 1$, we have an equivalent cost model for Lazy Leveling. However, our proposed K-LSM is the most flexible, as each level has an independent limit on the number of runs. Table 3 summarizes how K-LSM describes other common LSM variations:

## 5 The NOMINAL TUNING problem

In this section, we describe the classic tuning problem which involves finding the best configuration suited for a given workload without uncertainty. We examine the problem definition, algorithms to efficiently compute optimal configurations, and compare configurations across various designs of LSM trees.

### 5.1 NOMINAL TUNING Problem definition

Traditionally, designers focus on finding the configuration $\Phi^*$ that minimizes the total cost $C(\mathbf{w}, \Phi^*)$, for a given fixed workload $\mathbf{w}$. We call this problem the NOMINAL TUNING problem, which is defined as follows:

**Problem 1** (NOMINAL TUNING) Given a fixed workload $\mathbf{w}$, find the LSM tree configuration $\Phi_N$ such that

$$\Phi_N = \arg\min_{\Phi} C(\mathbf{w}, \Phi). \tag{10}$$

The problem described above captures the classic tuning paradigm of finding a system configuration that minimizes a cost model (describing I/O or latency) given a specific static workload and system environment. Therefore, each LSM design described in Table 3 has a different NOMINAL TUNING problem based on the form of the cost function. Prior tuning approaches either individually solve the NOMINAL TUNING problem solely for LSM data layouts [28, 53] (e.g., tiering or leveling) or memory allocation [26], but not simultaneously for both.

### 5.2 Solving a NOMINAL TUNING problem

To solve the NOMINAL TUNING problem, we utilize an off-the-shelf numerical solver. We opt to use the Sequential Least Square Quadratic Solver (SLSQP) implemented in Python and packaged under the SciPy library [84]. When choosing a data layout to optimize, we reduce the cost model to express the appropriate LSM design.

**Relaxing integer values**. Certain decision variables such as $T$ (size ratio) pose an issue as they are required to be integers as LSM trees cannot implement fractional size ratios. To keep the problem feasible, we relax the integer constraint for such decision variables and opt to take the ceiling of any feasible solution before deploying the tuning. In practice, this approach works well and leads to high-performance configurations.

### 5.3 Comparison of LSM strategies

In this section, we explore the optimal configurations for different designs described in Table 3 by solving the NOMINAL TUNING problem for each respective LSM design variation. We compare average I/Os per query to analyze the performances of different flexible designs.

**Experiment setup**. We adopt the following setting for system parameters: the database initially holds 10 billion entries, each of size 1 KB; a memory budget of 10 bits per element, or 10 GB in total divided among Bloom filter and write buffer; and a page size of 4 KB. It should be noted that the original Dostoevsky strategy uses Fluid LSM as an LSM design with fixed memory allocation, and only optimizes for the maximum number of runs for the upper levels, the lowest level, and size ratio while fixing memory. Therefore, while evaluating Dostoevsky we fix $m_{\text{filt}}$ to 10 bits per entry and $m_{\text{buf}}$ to 2 MB as in [28]. For all other design variations, we solve a NOMINAL TUNING problem that optimizes memory and

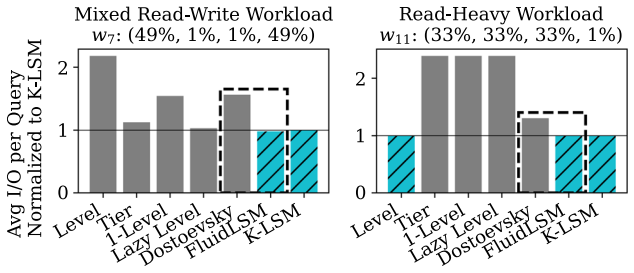



**Fig. 4** Throughput of different designs for fixed workloads. Hatched-cyan indicates the best performance. Note Dostoevsky uses the Fluid LSM design, but with fixed memory [28]

design while fixing other memory allocations such as fence pointers and the random access buffer.

**Flexible performance**. Figure 4 shows the average I/O performance of various tuning configurations normalized to K-LSM design across different workloads. We experiment with a mixed read-write ($\mathbf{w}_7$) and a read-heavy ($\mathbf{w}_{11}$) workload from the uncertainty benchmark, which is presented in detail in Sect. 7. Note that $\mathbf{w}_7$ would traditionally lead a designer to focus on tiering as writes make up a large portion of the workload, while $\mathbf{w}_{11}$ would suggest a leveling policy would be best. When solving for more flexible designs—in this instance K-LSM and Fluid LSM—we observe that the optimizer always produces the best tunings. Because $\mathbf{w}_{11}$ is a read-heavy distribution, the optimal configuration has a leveling policy, which is reinforced by observing that the optimal K-LSM and Fluid LSM designs chosen are equivalent leveling. For the balanced read-write workload $\mathbf{w}_7$, we see that flexible designs outperform traditional designs as the optimizer finely tunes the capacity per level to accommodate both reads and writes.

## 6 The ROBUST TUNING problem

In this section, we introduce the ROBUST TUNING problem, a variation of the NOMINAL TUNING problem that takes into consideration uncertainty in the workload. We give a precise definition of workload uncertainty and show how to compute high-performance configurations that minimize the expected cost of operation in the presence of this uncertainty.

### 6.1 ROBUST TUNING problem definition

The NOMINAL TUNING problem assumes perfect information about the workload before deploying 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 an observed workload may look like. Hence, the configuration obtained by solving Problem 1 may result in variable performance if the observed workload upon deployment varies greatly from the expected workload.

We capture this uncertainty by reformulating Problem 1 to take into account variability 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 $\Phi_R$ such that

$$\Phi_R = \arg\min_{\Phi} C(\hat{\mathbf{w}}, \Phi)$$
$$\text{s.t.,} \hat{\mathbf{w}} \in \mathcal{U}_{\mathbf{w}}. \tag{11}$$

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 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.

### 6.2 Solving the ROBUST TUNING problem

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 Eq. (11)) 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 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 [52] 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,

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

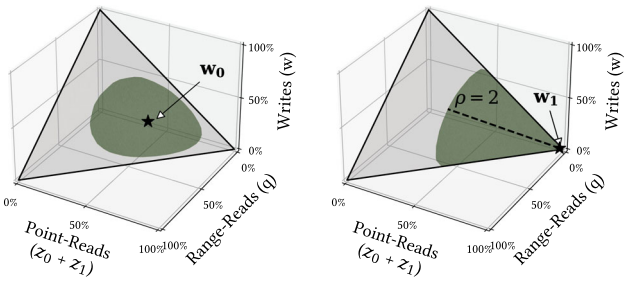



**Fig. 5** Workload uncertainty neighborhoods ($\mathcal{U}_{\mathbf{w}}$), denoted by the green shaded region, for two different expected workloads ($\mathbf{w}$) and $\rho$

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 $p^{\text{th}}$ power of differences in probabilities, which are extremely small values, and are not meaningful in this setting.

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

$$\mathcal{U}_{\mathbf{w}}^{\rho} = \{\hat{\mathbf{w}} \in \mathbb{R}^4 \mid \hat{\mathbf{w}} \geq 0, \hat{\mathbf{w}}^{\intercal}\mathbf{e} = 1, I_{KL}(\hat{\mathbf{w}}, \mathbf{w}) \leq \rho\}. \quad (12)$$

Here, $\rho$ 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 $\rho$, which intuitively defines the neighborhood around the expected workload. Figure 5 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 $\rho$, and the fact that all workloads are restricted to be probability distributions. In terms of notation, $\rho$ 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 $\rho$, the problem definition, as captured by Eq. (11), can be rewritten as follows:

$$\min_{\Phi} \max_{\hat{\mathbf{w}} \in \mathcal{U}_{\mathbf{w}}^{\rho}} \hat{\mathbf{w}}^{\intercal}\mathbf{c}(\Phi). \quad (13)$$

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 (13) can be rewritten by introducing an additional variable $\beta \in \mathbb{R}$, as follows:

$$\min_{\beta, \Phi} \quad \beta$$
$$\text{s.t., } \hat{\mathbf{w}}^{\intercal}\mathbf{c}(\Phi) \leq \beta \ \forall \hat{\mathbf{w}} \in \mathcal{U}_{\mathbf{w}}. \quad (14)$$

This reformulation allows us to remove the min max term in the objective from Eq. (13). The constraint in Eq. (14) can be equivalently expressed as,

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

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

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

where $\lambda$ and $\eta$ are the Lagrangian variables.

**Formulating the dual problem.** We can now express the dual objective as,

$$g(\lambda, \eta) = \max_{\hat{\mathbf{w}} \geq 0} \mathcal{L}(\hat{\mathbf{w}}, \lambda, \eta), \quad (15)$$

which we need to *minimize*.

Now we borrow the following result from [10],

**Lemma 1** ([10]) *A configuration $\Phi$ 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 Eq. (15)—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:

$$g(\lambda, \eta) = \eta + \rho\lambda + \lambda \sum_{i=1}^{k} \mathrm{w}_i \phi_{KL}^{*}\left(\frac{\mathbf{c}_i(\Phi) - \eta}{\lambda}\right). \quad (16)$$

In Eq. (16), $\phi_{KL}^{*}(.)$ 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 Sect. 3 – 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 Eq. (16) 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

$$\min_{\Phi,\lambda\geq 0,\eta} \left\{ \eta + \rho\lambda + \lambda \sum_{i=1}^{m} \mathrm{w}_i \phi_{KL}^{*} \left( \frac{c_i(\Phi) - \eta}{\lambda} \right) \right\}. \tag{17}$$

This minimization problem can be solved using the Sequential Least Squares Quadratic Programming solver (SLSQP) included in the popular Python optimization library SciPy [84]. Solving this problem outputs the values of the Lagrangian variables $\lambda$ and $\eta$ and most importantly the configuration $\Phi$ that optimizes the objective of the ROBUST TUNING problem—for input $\rho$. In terms of running time, the SLSQP solver outputs a robust tuning configuration for a given input in less than a second.

**Finding a value for $\rho$.** Since $\rho$ 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 $\rho$ using the following definition: that is, $\rho$ is set to be the largest KL-divergence between any observed workload and the corresponding workload average as described in Algorithm 1. 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 $\rho$ using the definition 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 $\rho$.

**Algorithm 1:** Calculating $\rho$ from historical workloads

**Input**: Set of historical workloads $\mathcal{W} = \{\mathbf{w}_1, \mathbf{w}_2, ..., \mathbf{w}_n\}$

1 $\bar{\mathbf{w}} \leftarrow \frac{1}{n} \cdot \sum_{\mathbf{w}_i \in \mathcal{W}} \mathbf{w}_i$

2 **return** $\arg\max_{\mathbf{w}_i \in \mathcal{W}} I_{KL}(\mathbf{w}_i, \bar{\mathbf{w}})$

## 7 Uncertainty benchmark

In this section, we describe the uncertainty benchmark that we use to evaluate the 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 tuning configurations for 15 expected workloads encompassing different proportions of query types. We catalog them into *uniform*, *unimodal*, *bimodal*, and *trimodal* categories based on 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 of 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 4.

**Table 4** Tested expected workloads

| 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% | |

**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 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 demonstrate that tuning with ENDURE achieves significant performance improvement using both a model-based analysis (Sect. 8), and an experimental study (Sect. 9).

## 8 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×. For brevity, when we provide a nominal tuning we are referring to the solution for the NOMINAL TUNING problem with tiering and leveling as the two design choices. Similarly, we use ENDURE and the robust tuning interchangeably to refer to the solution of the ROBUST TUNING problem which chooses between tiering and leveling. We show that ENDURE perfectly matches the nominal tuning when there is no uncertainty (i.e., 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 select uncertainty parameters.

### 8.1 Evaluation metrics

In this section, we provide definitions of metrics used to evaluate the performance of tunings.

**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 $\Phi_2$ w.r.t. another configuration $\Phi_1$ 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 $\Phi_2$ outperforms $\Phi_1$ 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 $\Phi$ 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 $\Phi$. A smaller value of this metric implies higher consistency in performance.

### 8.2 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 4, we obtain a single nominal tuning configuration ($\Phi_N$) by solving the NOMINAL TUNING problem. For 15 different values of $\rho$ in the range (0.0, 4.0) with a step size of 0.25, we obtain a set of robust tuning configurations ($\Phi_R$) 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 billion entries each of size 1 KB. The analysis presented in the following sections assumes a total available memory of 10 GB. For brevity, we present representative results corresponding to individual expected workloads and specific system parameters. We primarily focus on two workloads from Table 4, $\mathbf{w}_7$ which is a mixed read-write workload, and $\mathbf{w}_{11}$, which is a read-heavy workload. However, we exhaustively confirmed that changing these parameters does not qualitatively affect the outcomes of our experiment.

### 8.3 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 [84], we obtain both nominal and robust tunings with the runtime for the above experiment being less than 10 min.

**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 Fig. 6, we present performance comparisons between the robust and the nominal tunings for different values of uncertainty parameter $\rho$. 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 through-

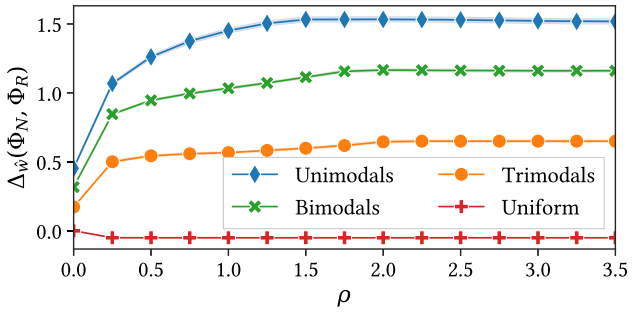



**Fig. 6** Average delta throughput $\Delta_{\hat{w}}(\Phi_N, \Phi_R)$ for each category of expected workload

put 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 $\rho$ covers a larger area of possible workloads than that same value would in a different workload as evident in Fig. 5. In this case, when tuned for high values of $\rho$, the robust tunings are unrealistically pessimistic and lose performance relative to the nominal tuning.

**Impact of tuning parameter $\rho$.** Next, we address the question—*how does the uncertainty tuning parameter $\rho$ impact the performance of the robust tunings?* In Fig. 7, we take a deep dive into the performance of robust tunings for an individual expected workload for different values of $\rho$. We observe that the robust tunings for $\rho = 0$ i.e., zero uncertainty, are very close to the nominal tunings. As the value of $\rho$ 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 Fig. 7 consists of just 1% writes. Hence, for low values of $\rho$, the robust tuning has a higher size ratio leading to shallower LSM trees to achieve good read performance. For higher values of $\rho$, the robust tunings anticipate an increasing percentage of write queries and hence limit the size ratio to achieve higher throughput.

In Fig. 8, we show the impact of tuning parameter $\rho$ on the throughput range. In Fig. 8a we plot a histogram of the nominal and robust throughputs for workload $\mathbf{w}_{11}$. As the value of $\rho$ 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 Fig. 8b, 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 for all $\hat{\mathbf{w}} \in \mathcal{B}$, but they have a more consistent performance (lower variance) compared to the nominal tunings.

**Choice of $\rho$.** Now, we address the question—*What is the appropriate choice for the value of uncertainty parameter $\rho$*? In Fig. 9, we explore the relationship between $\rho$ 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 tun-

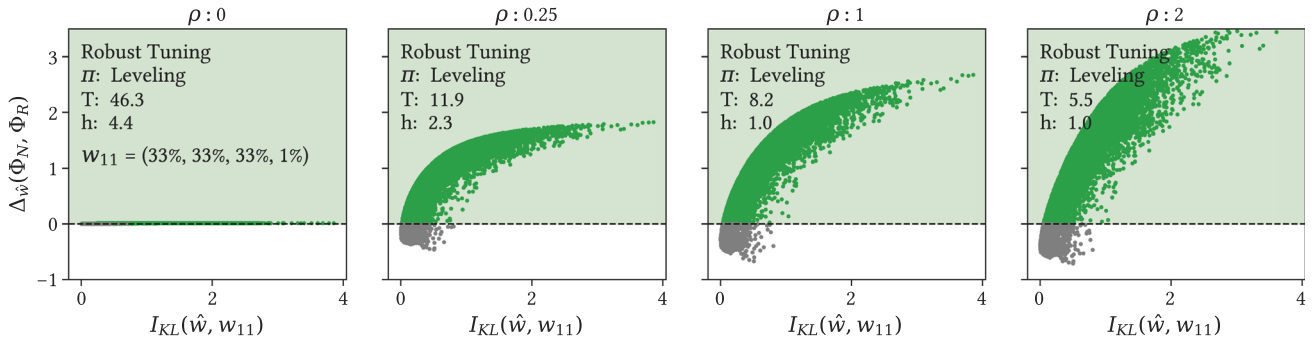



**Fig. 7** Impact of $\rho$ on normalized delta throughput $\Delta_{\hat{\mathbf{w}}}(\Phi_N, \Phi_R)$ for tunings with expected workload $\mathbf{w}_{11}$

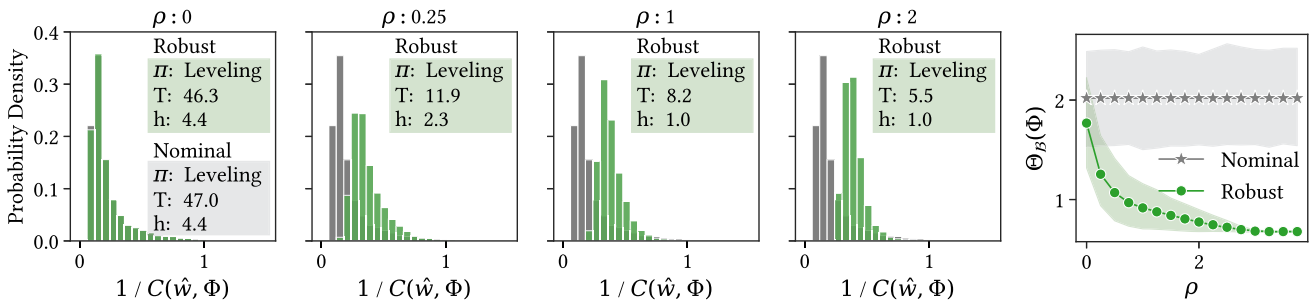



**Fig. 8** Impact of $\rho$ on throughput

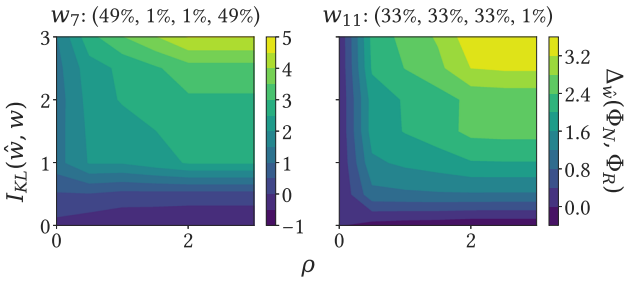



**Fig. 9** Effect on delta throughputs $\Delta_{\hat{\mathbf{w}}}(\Phi_N, \Phi_R)$ on selection of $\rho$ vs $I_{KL}(\hat{\mathbf{w}}, \mathbf{w})$

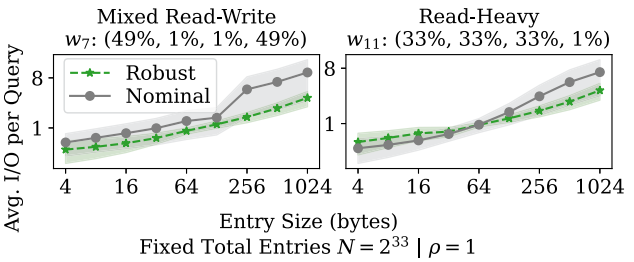



**Fig. 10** Tuning performance sensitivity to entry size

ings 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 $\rho$ in practice.

**Sensitivity analysis w.r.t. entry size**. Lastly, we take a look at the expected tuning performance w.r.t to system settings. Figure 10 shows average I/O, or single logical page accesses, per query over different entry sizes with the standard deviation highlighted around each line. Each data point corresponds to the average I/O per query for the optimal tuning for all workloads $\hat{\mathbf{w}} \in \mathcal{B}$. For our mixed read-write workload, we see that the ENDURE always performs better than the nominal tuning regardless of the entry size. When we tune with a read-heavy workload as the expected input, we observe that for lower entry sizes the nominal tuning produces a better tuning, however, at larger entry sizes ENDURE outperforms its nominal counterpart. Because the total number of entries is fixed, lower entry sizes cause the physical size of the database to be relatively small w.r.t. to the available memory budget. Hence, we observe the allocation between $m_{\text{filt}}$ and $m_{\text{buf}}$ does not play a major role in performance as the tree can be made relatively shallow. However, as the size of the database starts to increase and the memory budget becomes a smaller fraction of the database size, we observe the allocation between memory plays a larger role. This implies proper robust tunings play a larger role in constrained environments, where the available memory is a small fraction of the total database size.

## 9 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.

### 9.1 Experimental setup and 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 [32]. 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. For default RocksDB comparisons, we set a custom policy hook to match the default compaction policy of leveling. Additionally, RocksDB does not toggle on Bloom filters by default. In the interest of fair comparison, we add Bloom filters with the bits per element set to 10. Following the Monkey memory allocation scheme [26], we allocate different bits per element for Bloom filters per level. We note that turning off direct I/O improves read performance for any tuning deployed to RocksDB. However, to obtain an accurate count of block accesses we instead enable direct I/Os for both queries and compaction and disable the block cache. To obtain detailed insights about accesses, we present our findings with direct I/Os, however, our qualitative results remain unchanged with direct I/Os turned off. Lastly, portions of memory reserved for the fence pointer and max read buffer are fixed to their default values before performing any tuning for buffer size and Bloom filter memory.

ENDURE's *pipeline* Fig. 11 shows the workflow used for ENDURE and the following experiments. A workload descriptor (expected workload and uncertainty value $\rho$) is provided to ENDURE to create an uncertainty neighborhood centered around an expected workload. This description of workload uncertainty is then incorporated into the solver. In combination with the cost model, which uses the workload and system parameters as inputs, ENDURE outputs an expected performance profile and a robust tuning over various workloads in the uncertainty neighborhood. ENDURE then deploys

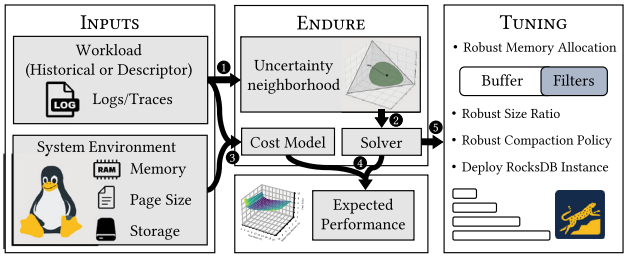



**Fig. 11** (1) Workload information is provided to ENDURE, establishing an uncertainty neighborhood centered on the expected workload. (2) This description of workload uncertainty is integrated into the solver. (3) The cost model receives both the workload and system parameter details. (4) Using the robust tuning from the solver and the cost model, an expected performance profile is generated. (5) The robust tuning is then deployed onto RocksDB

the robust tuning on a RocksDB instance where we execute workloads to measure performance.

**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 performance.

## 9.2 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 obtain consistent performance metrics, each instantiation of the database is initialized with the same 10 million unique 1 KB key-value pairs. 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 $\mathcal{B}$. Every sampled workload is executed throughout 200,000 queries to measure steady-state performance. This observation period is sufficient to capture spikes in performance and background compactions allowing us to record accurate performance numbers. We group sets of workloads into sequences and catalog them into one of six categories—*expected*, *empty read*, *non-empty read*, *read*, *range*, and *write*—based on the dominant query type. The *expected* session contains workloads with a KL-divergence less than 0.2 w.r.t. the expected workload used for tuning. For all other sessions, the dominant query type encompasses at least 80% of the total queries while the remaining queries may belong to any of the remaining types. When generating keys of the queries to run on the database, we ensure that non-empty point reads will query a key that exists, while empty point reads will 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, which on average read zero to two pages per level. Write queries consist of randomly generated keys that are distinct from the existing keys in the database. Similarly to Sect. 8, we present representative findings for an expected mixed read-write workload ($\mathbf{w}_7$) and an expected read-heavy workload ($\mathbf{w}_{11}$).

**Table 5** The system measured normalized delta throughputs $\Delta_{\mathbf{w}}(\Phi_N, \Phi_N)$ and their respective tunings for experiments on all expected workloads in $\mathcal{B}$ with an optimally selected $\rho$

| Expected Workload ($\mathbf{w}$) | $\Phi = (T, m_{\text{filt}}, \pi)$ $\Phi_N$ | $\Phi_R$ | $\Delta_{\mathbf{w}}(\Phi_N, \Phi_R)$ |
|---|---|---|---|
| $\mathbf{w}_0$ | (5.2, 3.5, **L**) | (5.1, 3.1, **L**) | 0.0 |
| $\mathbf{w}_1$ | (5.7, 9.4, **L**) | (5.0, 4.2, **L**) | 0.0 |
| $\mathbf{w}_2$ | (5.8, 5.3, **L**) | (5.0, 1.0, **L**) | 0.1 |
| $\mathbf{w}_3$ | (100, 0.0, **L**) | (5.4, 1.0, **L**) | 0.4 |
| $\mathbf{w}_4$ | (17, 3.2, **T**) | (4.6, 1.0, **L**) | 1.5 |
| $\mathbf{w}_5$ | (5.5, 8.8, **L**) | (5.1, 3.9, **L**) | 0.1 |
| $\mathbf{w}_6$ | (63, 4.8, **L**) | (8.2, 1.0, **L**) | 0.8 |
| $\mathbf{w}_7$ | (8.4, 8.2, **T**) | (3.4, 1.0, **L**) | 0.5 |
| $\mathbf{w}_8$ | (62, 0.0, **L**) | (8.0, 1.0, **L**) | 0.6 |
| $\mathbf{w}_9$ | (8.3, 6.9, **T**) | (3.3, 1.0, **L**) | 0.8 |
| $\mathbf{w}_{10}$ | (5.0, 0.0, **L**) | (5.0, 1.0, **L**) | 0.0 |
| $\mathbf{w}_{11}$ | (47, 4.7, **L**) | (11, 1.9, **L**) | 1.4 |
| $\mathbf{w}_{12}$ | (6.2, 8.1, **T**) | (2.8, 3.1, **L**) | 0.2 |
| $\mathbf{w}_{13}$ | (5.1, 3.5, **L**) | (5.0, 1.0, **L**) | −0.1 |
| $\mathbf{w}_{14}$ | (5.1, 0.0, **L**) | (5.0, 1.0, **L**) | −0.1 |

## 9.3 Experimental results

In this section, we replicate key insights from Sect. 8, evaluate system performance, and show that ENDURE scales with database size. We present detailed results for expected workloads $\mathbf{w}_7$ and $\mathbf{w}_{11}$. Table 5 summarizes the normalized delta throughputs $\Delta_{\mathbf{w}}(\Phi_N, \Phi_R)$ for all expected workload from $\mathcal{B}$.

**Cost of tuning**. We first solve for either the nominal or the robust tuning for every experiment. We note that solving either tuning problem 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

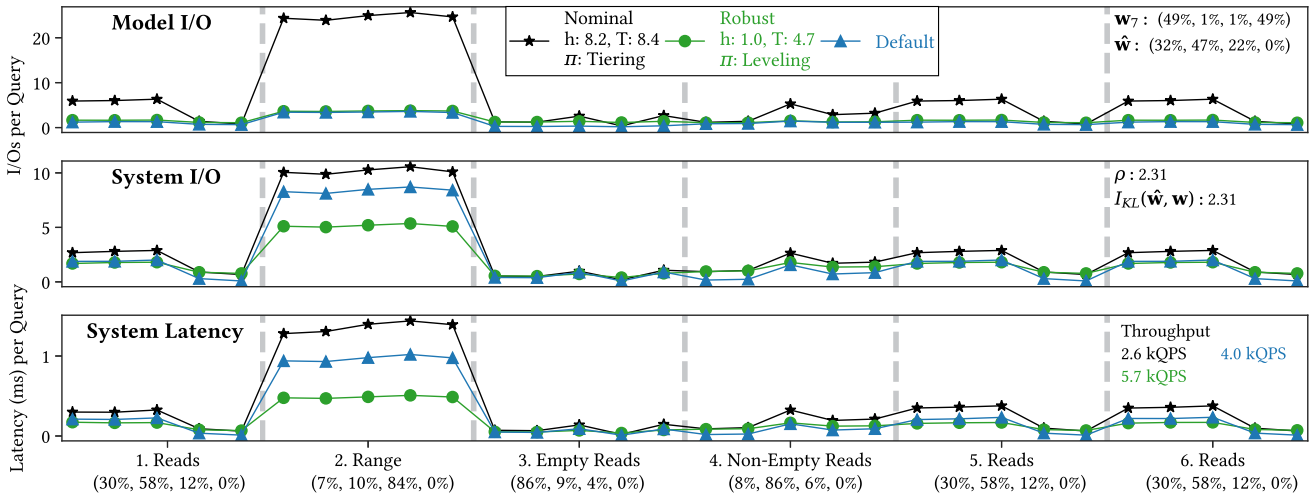



**Fig. 12** System and model performance for robust and nominal tunings in a read-only observed query sequence. Both tunings expected a mixed read-write workload. The tuning parameter $\rho$ (input uncertainty) matches the observed value of $I_{KL}(\hat{\mathbf{w}}, \mathbf{w}_7)$ (observed uncertainty). Each session contains the label and average workload

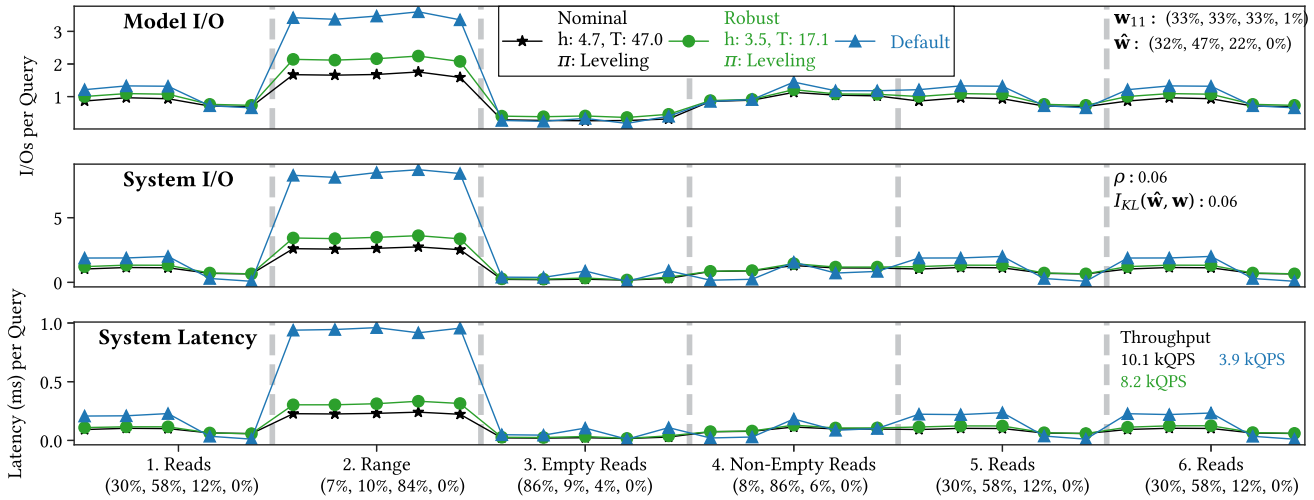



**Fig. 13** Read-only sequence where the observed workloads $\hat{\mathbf{w}}$ is close to the expected, hence $\rho$ and $I_{KL}(\hat{\mathbf{w}}, \mathbf{w}_{11})$ deviate. Both tunings expected a read-heavy workload

system-measured I/O match when considering read queries in Figs. 12 and 13. In both figures, we include the model-predicted I/Os per query (top), I/Os per query measured on the system (middle), and the system latency (bottom) for nominal, robust, and default configurations across different read sessions. Additionally, the total throughput numbers in queries per second are reported at the end of the 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 Fig. 12) 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. Lastly, we consider the default configuration as another tuning of RocksDB that does not take into account workload information. Therefore, in certain cases such as Fig. 12, it may outperform other configurations. This can be explained by the fact that the default configuration includes a larger reserve of memory for the Bloom filter, thereby allowing it to outperform the configurations that expect writes in the executed workload. However, in other cases such as Fig. 13, we see a large performance dropoff as both the nominal and robust configurations expect a high amount of reads and therefore tune their size ratios accordingly.

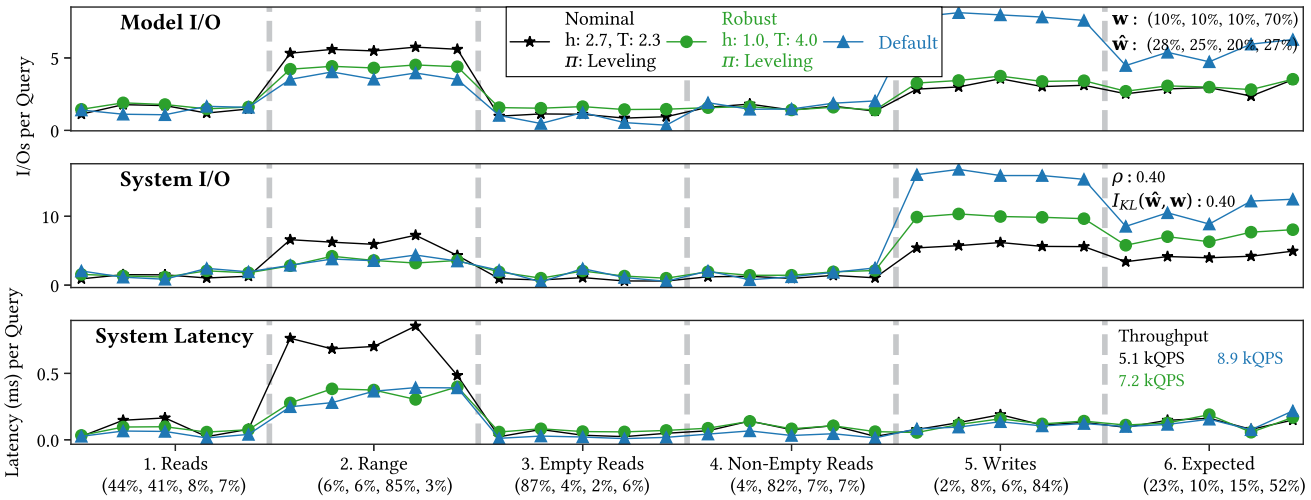



**Fig. 14** Sequence where $\rho$ and $I_{KL}(\hat{\mathbf{w}}, \mathbf{w})$ closely match. Both tunings expected a write-heavy workload. Performance fluctuates with writes as it modifies the tree

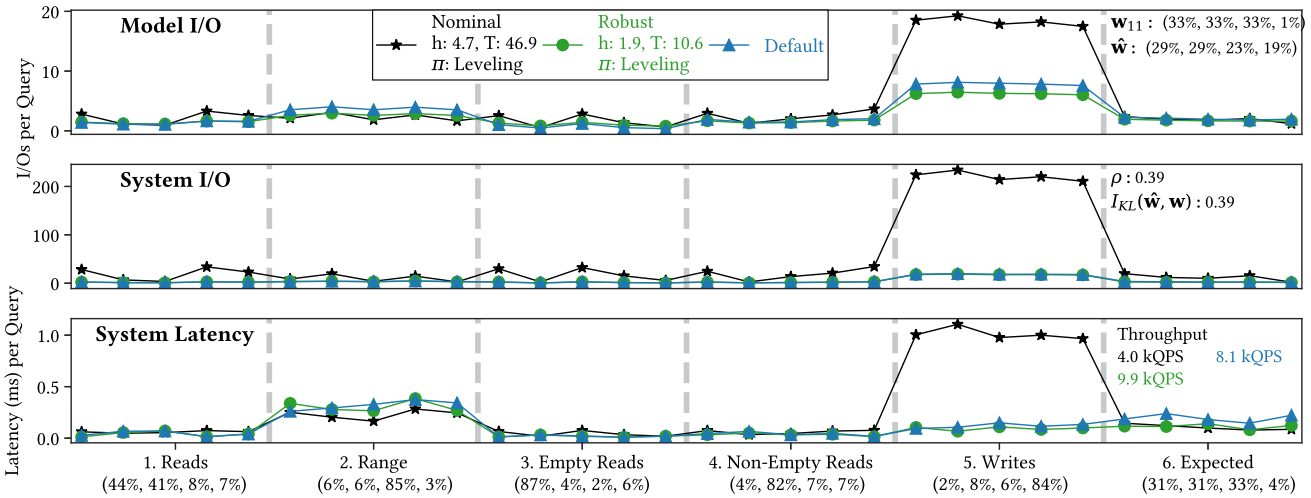



**Fig. 15** Sequence when $\rho$ and $I_{KL}(\hat{\mathbf{w}}, \mathbf{w}_{11})$ closely match. Both tunings expected a read-heavy workload. System I/O and latency show reductions of up to 90%

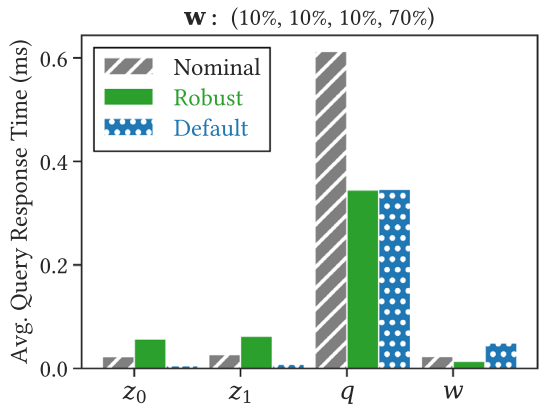



**Fig. 16** Breakdown of the query response time of each operation type for Fig. 14

**Write performance**. In the presence of writes in Fig. 15, 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 Fig. 15 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. Additionally, writes may appear instantaneous w.r.t. system latency as seen in Fig. 14 due to RocksDB assigning compactions to background threads. We observe that the default configuration starts to degrade in performance significantly as more writes are issued to the database. Figure 16 shows the breakdown across each operation type. As the database experiences more writes, the performance for the default configuration drops off, while both the nominal

tuning and robust configurations expect writes and experience a performance improvement. From the write session of Fig. 15, 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 huge levels, causing long stalls during compactions. Compare this to the robust tuning: the smaller size ratio creates a tree with more stable performance for both I/Os per query and query latency, leading to a higher overall throughput. Overall, we observe that the robust tuning reduces I/O and latency by up to 90%. Figures 12, 13, 14 and 15 confirm that our analytical model accurately captures the relative performance of different tunings.

**Robust outperforms nominal for properly selected $\rho$**. In the model evaluation (Fig. 9), we showed that robust tuning outperforms the nominal tuning in the presence of uncertainty for tuning parameter $\rho$ approximately greater than 0.2. This is further supported by all the system experiments described. Specifically, Figs. 12 and 15 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 $\rho$. Additionally, we present results for all expected workloads in Table 5. Each entry in the table summarizes the total throughput after running the same experimental setup presented in Fig. 15. 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 Fig. 13, the observed workloads are similar to the expected one ($I_{KL}(\hat{\mathbf{w}}, \mathbf{w}_{11}) < 0.2$), resulting in a latency increase of 20%.

**Balancing query times**. To determine how the tunings from ENDURE outperform the nominal tunings we analyze the query response times for each operation for an expected workload $\mathbf{w}$. We observe that robust tunings will provide lower performance for the queries that dominate the expected workload, however, as a tradeoff these tunings perform exceptionally well in unexpected operations. For example, Fig. 17 shows robust tuning performs worse in both range queries and empty point queries, however, in exchange we observe a significant decrease in the response time of write queries. Hence, if the executed workload contains writes, robust tuning will increase overall throughput, especially in the presence of write spikes.

**Workload skew**. To verify that ENDURE works across workload distributions, we break down the different query response times in Fig. 17. When the keys generated for the workload follow a Zipfian distribution, we see that the response time for non-empty read queries is significantly lowered. This is in part due to keys toward the top of the tree being repeated, therefore the query does not need to traverse further down the tree resulting in an increase in false

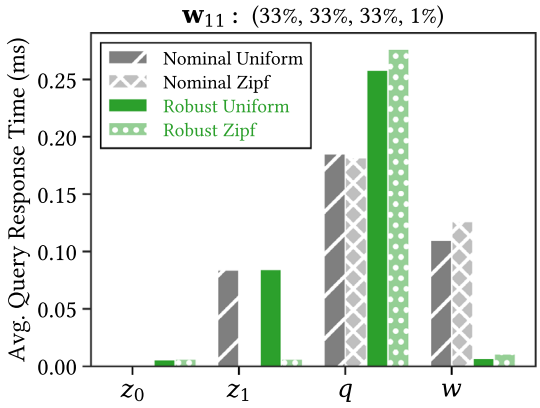



**Fig. 17** Query times for each operation (empty reads, non-empty reads, range reads, and writes) with an expected workload $\mathbf{w}_{11}$. Robust tunings were generated with $\rho = 1$

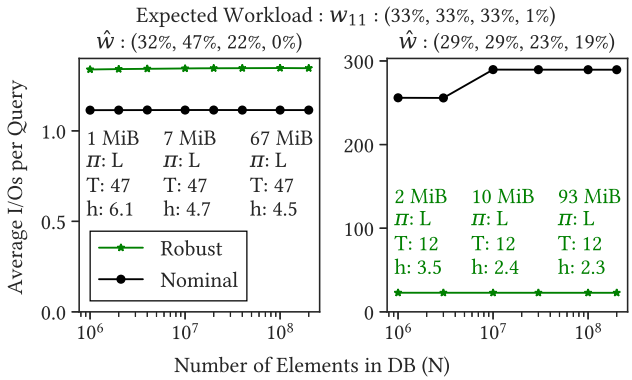



**Fig. 18** Impact of database size on performance. All tunings use the same expected workload $\mathbf{w}_{11}$ with executed workloads shown above each graph. Points at each power of 10 show $m_{buf}$ and the tuning $\Phi$ (L for leveling, T for tiering)

positives from the Bloom filter. Regardless, we observe the same patterns for uniform and Zipfian distributions; ENDURE tunings achieve a better tradeoff in performance for the dominant query types in the expected workload with that of other query types, thereby preventing large performance regressions.

**ENDURE scales with data size**. To verify that ENDURE scales, we repeat the previous experiments, while varying the size of the initial database. Each point in Fig. 18 is calculated based on a series of workload sessions similar to the ones presented in Figs. 13 (15) for the left (right) part of Fig. 18. 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 increases 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.

## 10 Robustness of flexible designs

In this section, we explore the nuanced differences between *flexibility* and *robustness* of LSM designs. Under an ideal scenario, where the expected workload is known a priori to tuning, the flexibility of K-LSM provides a high-performance benefit, however, this benefit vanishes in scenarios where the executed workload changes from the expected. This observation is in line with the intuition that ENDURE's robust tunings proactively compensate for potential changes in the workload distribution post-deployment, thereby providing better performance in situations where the executed workload differs from the expected.

### 10.1 Experiment design

To evaluate the robustness of all designs, we design an experiment to compare different optimal tunings for a subset of designs listed in Table 3. For each expected workload in the uncertainty benchmark in Table 4, we obtain a list of tunings by solving NOMINAL TUNING for various LSM data layouts. We also compute a robust tuning with input $\rho = 2$. Next, for every observed workload in $\mathcal{B}$, we calculate the KL-divergence w.r.t. the expected workload used to obtain each tuning and plot the average I/Os per query for this observed workload ($C(\hat{\mathbf{w}}, \Phi)$) versus the KL-divergence. We then execute the initial expected workload for all designs, reset the database to the initial state, and then progressively repeat this process for workloads further away from the expected workload. Similar to Sect. 5.3, we adopt the following setting for system parameters: the database initially holds 10 billion with each entry at 1 KB; page size is 4 KB; and memory budget is set to 10 bits per element or a total of 10 GB.

### 10.2 Comparison results

Figure 19 shows the average I/O per query for various LSM models. Note that for $\mathbf{w}_{11}$, the performance lines for K-LSM, Fluid LSM, and ENDURE's nominal tuning slightly overlap, as the configurations are the same. The same occurs for K-LSM and Fluid LSM on $\mathbf{w}_7$. When evaluating Dostoevsky, we fix memory allocation such that the buffer size is kept at 2 MB as per convention, while the remaining memory is delegated to storing Bloom filters [28, 69].

**Robustness of various LSM designs**. It should be noted that in instances where the observed workload closely matches the expected workload, ENDURE's robust tunings underperform. This is consistent with previous experiments and the intuition that robust tunings proactively compensate for potential changes in the observed workload distribution. As the observed workload drifts further from the expected, robust tunings maintain consistent performance while other tunings show a steady increase in the average number of I/Os. This observation can be attributed to the selected tuning. For example, with $\mathbf{w}_7$ models such as Dostoevsky ($T = 47$, all $K_i = 1$) and K-LSM ($m_{\text{filt}} = 4.4$ bits per entry, $T = 48$, all $K_i = 1$) optimally selects larger size ratios with effectively leveling policies to accommodate for the expected high amount of writes. In contrast, the robust tuning ($T = 9$, $\pi = L$) selects a size ratio that performs reasonably well in comparison, however, the selected size ratio is small enough to accommodate a large shift to reads.

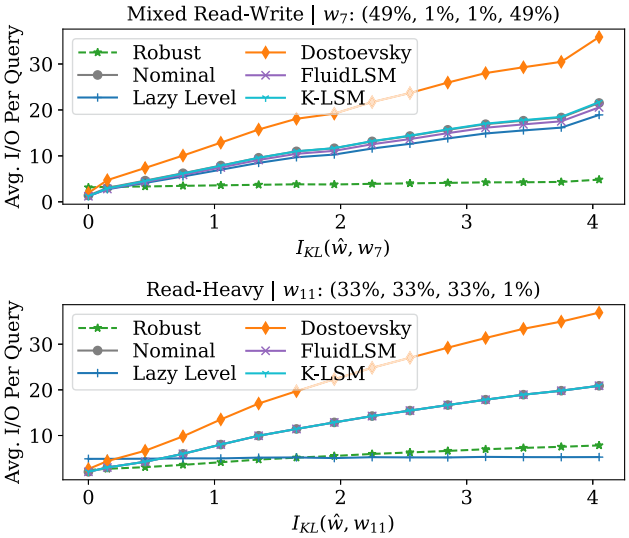



**Fig. 19** The cost of each LSM model as the observed workload $\hat{\mathbf{w}}$ drifts away from the expected workload $\mathbf{w}$

In the presence of workload drifts, we observe that most models, except Lazy Leveling, experience a performance degradation similar to ENDURE's nominal tuning. The optimal tuning of Lazy Leveling ($m_{\text{filt}} = 4.6$ bits per entry, $T = 2$) performs robustly when tuned for a read-heavy workload ($\mathbf{w}_{11}$). It should be noted that the optimizer selects a size ratio $T = 2$ that enables Lazy Leveling to accommodate an increase in writes, as merge operations are relatively cheap. Furthermore, increasing the size ratio any further could lead to the creation of upper levels that follow a tiering policy, thereby degrading the performance.

## 11 Discussion

In this section, we discuss the key insights gained from benchmarking and testing ENDURE.

**Robustness is all you need**. When deploying LSM trees, it is evident that tuning with some knowledge about the workload can improve performance, but accounting for uncertainty in

the tuning process can provide an even greater benefit for performance in the long run. To support this, in Sect. 9.3, we show that the cost model can accurately predict the empirical measurements. Then 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 5, 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.

**Robustness is not inherent**. As evident in Fig. 19, the final takeaway when evaluating robust tunings compared to optimal tuning of other flexible models is that *robustness is not inherent to a model and must be considered in the tuning process*. We observe that flexible models may provide better initial performance, however, only ENDURE, which explicitly accounts for workload uncertainty in the tuning process, performs well w.r.t. to a changing workload. While other models may exhibit some degree of robust performance in specific and limited scenarios, only the robust tuning consistently performs well in the presence of workload drift across all different expected workloads. Based on our analytical and empirical results, we recommend that robust tuning should always be employed when tuning an LSM tree unless the future workload distribution is known with an absolute certainty.

**Limitations**. One of the key challenges during the evaluation of tuning configurations in the presence of uncertainty is in measuring steady-state performance. Background compactions create variability in performance requiring longer database testing sessions to see accurate performance numbers. To observe trends across multiple tunings we had to strike a balance between exhaustive testing and runtime. Using off-the-shelf optimizers, such as the SLSQP solver from SciPy mentioned in Sect. 5, present restrictions in terms of the complexity of designs that we can optimally tune. Numerical solvers are sensitive to hyperparameters such as starting conditions and step size. Therefore, tuning performance can greatly vary based on the correct initialization of such hyperparameters. Furthermore, the stability and accuracy of numerical solvers suffer with an increase in the number of decision variables. We observed that when solving the ROBUST TUNING problem for the most flexible designs, the combination of hyperparameter sensitivity and numerical instability with additional decision variables leads to suboptimal solutions.

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 surrogate.

## 12 Related work

Database tuning is a notoriously hard problem, however, there has been a plethora of recent research. We provide a discussion of related works around the field of data systems tuning.

**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, 83, 91], and periodic online workload analysis [16, 74–76] 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, 63, 78, 80, 81]. 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 and 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, 77]. 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, 59, 65] 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 suboptimal decisions when there is significant workload drift. Cliffguard [60] 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 Eq. (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 an extensive hyperparameter search.

## 13 Conclusion

In this work, we explored the impact of workload uncertainty and LSM design flexibility on the performance of LSM tree databases. Based on our explorations, we introduce ENDURE—a robust tuning paradigm that recommends robust designs to mitigate performance degradation under scenarios of deviating workloads. We showed that in the presence of uncertainty, ENDURE increases database throughput compared to standard tunings by up to 5×. Furthermore, we proposed a unified LSM design with an associated flexible cost model that can express multiple LSM data layout designs, and provide evidence that our cost model closely matches the behavior measured on a database system. We used this cost model to analyze the robustness of flexible models and provide evidence that *robustness is not inherent* to a particular design, rather it must be an important consideration during the tuning process. 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 without resorting to expensive database experiments.

**Acknowledgements** We thank Anwesha Saha and Sakshi Sharma for their help in the experimental analysis and the anonymous reviewers for their valuable feedback. This work is partially funded by an IBM Ph.D. Fellowship, a Red Hat Incubation Award, a Meta gift, and NSF Grants No. IIS-1813406, No. IIS-1908510, and No. IIS-2144547.



## References

1. Absalyamov, I., Carey, M.J., Tsotras, V.J.: Lightweight cardinality estimation in LSM-based systems. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 841–855 (2018). https://doi.org/10.1145/3183713.3183761
2. Agrawal, S., Chaudhuri, S., Kollár, L., Marathe, A.P., Narasayya, V.R., Syamala, M.: Database tuning advisor for Microsoft SQL server 2005. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 1110–1121 (2004). https://doi.org/10.1145/1066157.1066292
3. Agrawal, S., Chaudhuri, S., Narasayya, V.R.: Automated selection of materialized views and indexes in SQL databases. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 496–505 (2000). https://doi.org/10.5555/645926.671701
4. Aken, D.V., Pavlo, A., Gordon, G.J., Zhang, B.: Automatic database management system tuning through large-scale machine learning. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 1009–1024 (2017). https://doi.org/10.1145/3035918.3064029
5. Alkowaileet, W.Y., Alsubaiee, S., Carey, M.J.: An LSM-based tuple compaction framework for apache AsterixDB. Proc. VLDB Endow. **13**(9), 1388–1400 (2020). https://doi.org/10.14778/3397230.3397236
6. Alsubaiee, S., Altowim, Y., Altwaijry, H., Behm, A., Borkar, V.R., Bu, Y., Carey, M.J., Cetindil, I., Cheelangi, M., Faraaz, K., Gabrielova, E., Grover, R., Heilbron, Z., Kim, Y.S., Li, C., Li, G., Ok, J.M., Onose, N., Pirzadeh, P., Tsotras, V.J., Vernica, R., Wen, J., Westmann, T.: AsterixDB: a scalable, open source BDMS. Proc. VLDB Endow. **7**(14), 1905–1916 (2014). https://doi.org/10.14778/2733085.2733096
7. Apache: Apache HBase (2013). http://hbase.apache.org/
8. Apache: Cassandra. http://cassandra.apache.org (2023)
9. Athanassoulis, M., Bøgh, K.S., Idreos, S.: Optimal column layout for hybrid workloads. Proc. VLDB Endow. **12**(13), 2393–2407 (2019). https://doi.org/10.14778/3358701.3358707
10. Ben-Tal, A., den Hertog, D., Waegenaere, A.D., Melenberg, B., Rennen, G.: Robust solutions of optimization problems affected by uncertain probabilities. Manag. Sci. **59**(2), 341–357 (2013). https://doi.org/10.1287/mnsc.1120.1641
11. Ben-Tal, A., Nemirovski, A.: Robust convex optimization. Math. Oper. Res. **23**(4), 769–805 (1998). https://doi.org/10.1287/moor.23.4.769
12. Bertsimas, D., Nohadani, O., Teo, K.M.: Robust optimization for unconstrained simulation-based problems. Oper. Res. **58**(1), 161–178 (2010). https://doi.org/10.1287/opre.1090.0715

13. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970). https://doi.org/10.1145/362686.362692
14. Bortnikov, E., Braginsky, A., Hillel, E., Keidar, I., Sheffi, G.: Accordion: better memory organization for LSM key-value stores. Proc. VLDB Endow. **11**(12), 1863–1875 (2018). https://doi.org/10.14778/3229863.3229873
15. Bruno, N., Chaudhuri, S.: Automatic physical database tuning. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 227–238 (2005). https://doi.org/10.1145/1066157.1066184
16. Bruno, N., Chaudhuri, S.: To tune or not to Tune? A lightweight physical design Alerter. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 499–510 (2006). https://doi.org/10.5555/1182635.1164171
17. Cao, Z., Dong, S., Vemuri, S., Du, D.H.C.: Characterizing, modeling, and benchmarking RocksDB key-value workloads at facebook. In: Proceedings of the USENIX Conference on File and Storage Technologies (FAST), pp. 209–223 (2020). https://doi.org/10.5555/3386691.3386712
18. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 205–218 (2006). https://doi.org/10.5555/1267308.1267323
19. Chaudhuri, S., Dageville, B., Lohman, G.M.: Self-managing technology in database management systems. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), p. 1243 (2004). https://doi.org/10.1016/B978-012088469-8.50116-9
20. Chaudhuri, S., Narasayya, V.R.: An efficient cost-driven index selection tool for Microsoft SQL server. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 146–155 (1997). https://doi.org/10.5555/645923.673646
21. Chaudhuri, S., Narasayya, V.R.: AutoAdmin 'What-if' index analysis utility. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 367–378 (1998). https://doi.org/10.1145/276304.276337
22. Chaudhuri, S., Weikum, G.: Foundations of automated database tuning. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 964–965 (2005). https://doi.org/10.1145/1066157.1066305
23. Chohan, N., Castillo, C., Spreitzer, M., Steinder, M., Tantawi, A.N., Krintz, C.: See spot run: using spot instances for MapReduce workflows. In: Proceedings of USENIX Workshop on Hot Topics in Cloud Computing (HotCloud) (2010). https://doi.org/10.5555/1863103.1863110
24. Dageville, B., Das, D., Dias, K., Yagoub, K., Zait, M., Ziauddin, M.: Automatic SQL tuning in oracle 10g. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 1098–1109 (2004). https://doi.org/10.5555/1316689.1316784
25. Dash, D., Polyzotis, N., Ailamaki, A.: CoPhy: a scalable, portable, and interactive index advisor for large workloads. Proc. VLDB Endow. **4**(6), 362–372 (2011). https://doi.org/10.14778/1978665.1978668
26. Dayan, N., Athanassoulis, M., Idreos, S.: Monkey: optimal navigable key-value store. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 79–94 (2017). https://doi.org/10.1145/3035918.3064054
27. Dayan, N., Athanassoulis, M., Idreos, S.: Optimal bloom filters and adaptive merging for LSM-trees. ACM Trans. Database Syst. **43**(4), 16:1-16:48 (2018). https://doi.org/10.1145/3276980
28. Dayan, N., Idreos, S.: 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, pp. 505–520 (2018). https://doi.org/10.1145/3183713.3196927
29. Dayan, N., Idreos, S.: The log-structured merge-bush and the wacky continuum. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 449–466 (2019). https://doi.org/10.1145/3299869.3319903
30. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. ACM SIGOPS Oper. Syst. Rev. **41**(6), 205–220 (2007). https://doi.org/10.1145/1323293.1294281
31. Dong, S., Callaghan, M., Galanis, L., Borthakur, D., Savor, T., Strum, M.: Optimizing space amplification in RocksDB. In: Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR) (2017). https://www.cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf
32. Facebook: RocksDB (2021). https://github.com/facebook/rocksdb
33. Galante, G., Bona, L.C.E.D.: A survey on cloud computing elasticity. In: Proceedings of the IEEE International Conference on Utility and Cloud Computing (UCC), pp. 263–270 (2012). https://doi.org/10.1109/UCC.2012.30
34. Google: LevelDB (2021). https://github.com/google/leveldb/
35. Graefe, G., Kuno, H.: Self-selecting, self-tuning, incrementally optimized indexes. In: Proceedings of the International Conference on Extending Database Technology (EDBT), pp. 371–381 (2010). https://doi.org/10.1145/1739041.1739087
36. Graefe, G., Kuno, H.A.: Adaptive indexing for relational keys. In: Proceedings of the IEEE International Conference on Data Engineering Workshops (ICDEW), pp. 69–74 (2010). https://doi.org/10.1109/ICDEW.2010.5452743
37. Hayes, B.: Cloud computing. Commun. ACM **51**(7), 9–11 (2008). https://doi.org/10.1145/1364782.1364786
38. Herbst, N.R., Kounev, S., Reussner, R.H.: Elasticity in cloud computing: what it is, and what it is not. In: Proceedings of the International Conference on Autonomic Computing (ICAC), pp. 23–27 (2013). https://www.usenix.org/conference/icac13/technical-sessions/presentation/herbst
39. Holze, M., Haschimi, A., Ritter, N.: Towards workload-aware self-management: predicting significant workload shifts. Proceedings of the IEEE International Conference on Data Engineering (ICDE) pp. 111–116 (2010). https://doi.org/10.1109/ICDEW.2010.5452738
40. Huang, G., Cheng, X., Wang, J., Wang, Y., He, D., Zhang, T., Li, F., Wang, S., Cao, W., Li, Q.: 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, pp. 651–665 (2019). https://doi.org/10.1145/3299869.3314041
41. Huynh, A., Chaudhari, H.A., Terzi, E., Athanassoulis, M.: Endure: a robust tuning paradigm for LSM trees under workload uncertainty. Proceedings of the VLDB Endowment **15**(8), 1605–1618 (2022). 0.14778/3529337.3529345
42. Idreos, S., Dayan, N., Qin, W., Akmanalp, M., Hilgard, S., Ross, A., Lennon, J., Jain, V., Gupta, H., Li, D., Zhu, Z.: 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) (2019). https://www.cidrdb.org/cidr2019/papers/p143-idreos-cidr19.pdf
43. Idreos, S., Kersten, M.L., Manegold, S.: Database cracking. In: Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR) (2007). https://www.cidrdb.org/cidr2007/papers/cidr07p07.pdf
44. Idreos, S., Kraska, T.: 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 (2019). https://doi.org/10.1145/3299869.3314034

45. Idreos, S., Zoumpatianos, K., Athanassoulis, M., Dayan, N., Hentschel, B., Kester, M.S., Guo, D., Maas, L.M., Qin, W., Wasay, A., Sun, Y.: The periodic table of data structures. IEEE Data Eng. Bull. **41**(3), 64–75 (2018)
46. Idreos, S., Zoumpatianos, K., Hentschel, B., Kester, M.S., Guo, D.: 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, pp. 535–550 (2018). https://doi.org/10.1145/3183713.3199671
47. Influxdata: in-memory indexing and the time-structured merge tree (TSM) (2021). https://docs.influxdata.com/influxdb/v1.8/concepts/storage_engine/
48. Jagadish, H.V., Narayan, P.P.S., Seshadri, S., Sudarshan, S., Kanneganti, R.: Incremental organization for data recording and warehousing. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 16–25 (1997). https://doi.org/10.5555/645923.671013
49. Kim, T., Behm, A., Blow, M., Borkar, V., Bu, Y., Carey, M.J., Hubail, M., Jahangiri, S., Jia, J., Li, C., Luo, C., Maxon, I., Pirzadeh, P.: Robust and efficient memory management in Apache AsterixDB. Softw. Pract. Exp. **50**(7), 1114–1151 (2020). https://doi.org/10.1002/spe.2799
50. Kim, Y.S., Kim, T., Carey, M.J., Li, C.: A comparative study of log-structured merge-tree-based spatial indexes for big data. In: Proceedings of the IEEE International Conference on Data Engineering (ICDE), pp. 147–150 (2017). https://doi.org/10.1109/ICDE.2017.61
51. Kraska, T.: Towards instance-optimized data systems. Proc. VLDB Endow. **14**(12), 3222–3232 (2021). https://doi.org/10.14778/3476311.3476392
52. Kullback, S., Leibler, R.A.: On information and sufficiency. Ann. Math. Stat. **22**(1), 79–86 (1951). https://doi.org/10.1214/aoms/1177729694
53. Luo, C.: Breaking down memory walls in LSM-based storage systems. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 2817–2819 (2020). https://doi.org/10.1145/3318464.3384399
54. Luo, C., Carey, M.J.: On performance stability in LSM-based storage systems. Proc. VLDB Endow. **13**(4), 449–462 (2019). https://doi.org/10.14778/3372716.3372719
55. Luo, C., Carey, M.J.: Breaking down memory walls: adaptive memory management in LSM-based storage systems. Proc. VLDB Endow. **14**(3), 241–254 (2020). https://doi.org/10.5555/3430915.3442425
56. Luo, C., Carey, M.J.: LSM-based storage techniques: a survey. VLDB J. **29**(1), 393–418 (2020). https://doi.org/10.1007/s00778-019-00555-y
57. Luo, C., Tözün, P., Tian, Y., Barber, R., Raman, V., Sidle, R.: Umzi: unified multi-zone indexing for large-scale HTAP. In: Proceedings of the International Conference on Extending Database Technology (EDBT), pp. 1–12 (2019). https://doi.org/10.5441/002/edbt.2019.02
58. Luo, S., Chatterjee, S., Ketsetsidis, R., Dayan, N., Qin, W., Idreos, S.: Rosetta: a robust space-time optimized range filter for key-value stores. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 2071–2086 (2020). https://doi.org/10.1145/3318464.3389731
59. Ma, L., Aken, D.V., Hefny, A., Mezerhane, G., Pavlo, A., Gordon, G.J.: Query-based workload forecasting for self-driving database management systems. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 631–645 (2018). https://doi.org/10.1145/3183713.3196908
60. Mozafari, B., Goh, E.Z.Y., Yoon, D.Y.: CliffGuard: a principled framework for finding robust database designs. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 1167–1182 (2015). https://doi.org/10.1145/2723372.2749454
61. O'Neil, P.E., Cheng, E., Gawlick, D., O'Neil, E.J.: The log-structured merge-tree (LSM-tree). Acta Informatica **33**(4), 351–385 (1996). https://doi.org/10.1007/s002360050048
62. Özcan, F., Tian, Y., Tözün, P.: Hybrid transactional/analytical processing: a survey. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 1771–1775 (2017). https://doi.org/10.1145/3035918.3054784
63. Papadomanolakis, S., Ailamaki, A.: AutoPart: automating schema design for large scientific databases using data partitioning. In: Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM), p. 383 (2004). https://doi.org/10.1109/SSDBM.2004.19
64. Papon, T.I., Athanassoulis, M.: A parametric I/O model for modern storage devices. In: Proceedings of the International Workshop on Data Management on New Hardware (DAMON) (2021). https://doi.org/10.1145/3465998.3466003
65. Pavlo, A., Angulo, G., Arulraj, J., Lin, H., Lin, J., Ma, L., Menon, P., Mowry, T.C., Perron, M., Quah, I., Santurkar, S., Tomasic, A., Toor, S., Aken, D.V., Wang, Z., Wu, Y., Xian, R., Zhang, T.: Self-driving database management systems. In: Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR) (2017)
66. Pezzini, M., Feinberg, D., Rayner, N., Edjlali, R.: Hybrid transaction/analytical processing will foster opportunities for dramatic business innovation. https://www.gartner.com/doc/2657815/ (2014)
67. Ren, K., Zheng, Q., Arulraj, J., Gibson, G.: SlimDB: a space-efficient key-value storage engine for semi-sorted data. Proc. VLDB Endow. **10**(13), 2037–2048 (2017). https://doi.org/10.14778/3151106.3151108
68. Research, G.V.: 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). https://www.grandviewresearch.com/industry-analysis/private-cloud-server-market
69. RocksDB: RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide (2021)
70. Sarkar, S., Athanassoulis, M.: Dissecting, designing, and optimizing LSM-based data stores. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 2489–2497 (2022). https://doi.org/10.1145/3514221.3522563
71. Sarkar, S., Dayan, N., Athanassoulis, M.: The LSM design space and its read optimizations. In: Proceedings of the IEEE International Conference on Data Engineering (ICDE) (2023). https://doi.org/10.1109/ICDE55515.2023.00273
72. Sarkar, S., Papon, T.I., Staratzis, D., Athanassoulis, M.: Lethe: a tunable delete-aware LSM engine. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 893–908 (2020). https://doi.org/10.1145/3318464.3389757
73. Sarkar, S., Staratzis, D., Zhu, Z., Athanassoulis, M.: Constructing and analyzing the LSM compaction design space. Proc. VLDB Endow. **14**(11), 2216–2229 (2021). https://doi.org/10.14778/3476249.3476274
74. Schnaitter, K., Abiteboul, S., Milo, T., Polyzotis, N.: COLT: continuous on-line database tuning. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 793–795 (2006). https://doi.org/10.1145/1142473.1142592
75. Schnaitter, K., Abiteboul, S., Milo, T., Polyzotis, N.: On-line index selection for shifting workloads. In: Proceedings of the IEEE International Conference on Data Engineering Workshops (ICDEW), pp. 459–468 (2007). https://doi.org/10.1109/ICDEW.2007.4401029

76. Schnaitter, K., Polyzotis, N.: Semi-automatic index tuning. Proc. VLDB Endow. **5**(5), 478–489 (2012). https://doi.org/10.14778/2140436.2140444
77. Schuhknecht, F.M., Dittrich, J., Linden, L.: Adaptive adaptive indexing. In: Proceedings of the IEEE International Conference on Data Engineering (ICDE), pp. 665–676 (2018). https://doi.org/10.1109/ICDE.2018.00066
78. Serafini, M., Taft, R., Elmore, A.J., Pavlo, A., Aboulnaga, A., Stonebraker, M.: Clay: fine-grained adaptive partitioning for general database schemas. Proc. VLDB Endow. **10**(4), 445–456 (2016). https://doi.org/10.14778/3025111.3025125
79. Shasha, D.E., Bonnet, P.: Database tuning: principles, experiments, and troubleshooting techniques. In: Proceedings of the International Conference on Very Large Data Bases (VLDB) (2002). https://doi.org/10.1145/1024694.1024720
80. Sun, L., Franklin, M.J., Krishnan, S., Xin, R.S.: Fine-grained partitioning for aggressive data skipping. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 1115–1126 (2014). https://doi.org/10.1145/2588555.2610515
81. Sun, L., Franklin, M.J., Wang, J., Wu, E.: Skipping-oriented Partitioning for Columnar Layouts. Proceedings of the VLDB Endowment **10**(4), 421–432 (2016). https://doi.org/10.14778/3025111.3025123
82. Tarkoma, S., Rothenberg, C.E., Lagerspetz, E.: Theory and practice of bloom filters for distributed systems. IEEE Commun. Surv. Tutorials **14**(1), 131–155 (2012). https://doi.org/10.1109/SURV.2011.031611.00024
83. Valentin, G., Zuliani, M., Zilio, D.C., Lohman, G.M., Skelley, A.: DB2 Advisor: an optimizer smart enough to recommend its own indexes. In: Proceedings of the IEEE International Conference on Data Engineering (ICDE), pp. 101–110 (2000). https://doi.org/10.1109/ICDE.2000.839397
84. Virtanen, P., Gommers, R., Oliphant, T.E., Haberland, M., Reddy, T., et al.: Cournapeau D: SciPy 1.0: fundamental algorithms for scientific computing in python. Nat. Methods **17**, 261–272 (2020). https://doi.org/10.1038/s41592-019-0686-2
85. Wikipedia contributors: Bloom filter—Wikipedia, the free encyclopedia (2021). https://en.wikipedia.org/w/index.php?title=Bloom_filter. Accessed 8 June 2021
86. WiredTiger: Wiredtiger (2021). https://github.com/wiredtiger/wiredtiger
87. Wolski, R., Brevik, J., Chard, R., Chard, K.: Probabilistic guarantees of execution duration for Amazon spot instances. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 18:1–18:11 (2017). https://doi.org/10.1145/3126908.3126953
88. Yang, L., Wu, H., Zhang, T., Cheng, X., Li, F., Zou, L., Wang, Y., Chen, R., Wang, J., Huang, G.: Leaper: a learned prefetcher for cache invalidation in LSM-tree based storage engines. Proc. VLDB Endow. **13**(11), 1976–1989 (2020). https://doi.org/10.14778/3407790.3407803
89. Zhang, H., Lim, H., Leis, V., Andersen, D.G., Kaminsky, M., Keeton, K., Pavlo, A.: SuRF: practical range query filtering with fast succinct tries. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 323–336 (2018). https://doi.org/10.1145/3183713.3196931
90. Zhang, T., Wang, J., Cheng, X., Xu, H., Yu, N., Huang, G., Zhang, T., He, D., Li, F., Cao, W., Huang, Z., Sun, J.: FPGA-accelerated compactions for LSM-based key-value store. In: Proceedings of the USENIX Conference on File and Storage Technologies (FAST), pp. 225–237 (2020). https://doi.org/10.5555/3386691.3386713
91. Zilio, D.C., Rao, J., Lightstone, S., Lohman, G.M., Storm, A., Garcia-Arellano, C., Fadden, S.: DB2 design Advisor: integrated automatic physical database design. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 1087–1097 (2004). https://doi.org/10.5555/1316689.1316783