AXE: A Task Decomposition Approach to Learned LSM Tuning

Andy Huynh
Boston University
ndhuynh@bu.edu

Anwesha Saha
Boston University
anwesha@bu.edu

Harshal A. Chaudhari
Boston University
harshal@bu.edu

Manos Athanassoulis
Boston University
mathan@bu.edu

ABSTRACT

Log-Structured Merge (LSM) trees are used as the data structure of choice for key-value stores supporting a wide variety of applications. A common challenge for LSM-based systems is tuning them effectively, particularly as the complexity and number of tuning knobs increase. Prior work relies on expert-created cost models and expert-configured numerical solvers to produce high-quality tunings; however, these methods do not address tuning multiple instances at scale for various execution environments. On the other hand, using iterative learning, such as Bayesian Optimization (BO), relaxes the requirements for domain expertise and provides generalizability; however, it comes at a high cost, as it involves learning directly from database executions at deployment time. Furthermore, both approaches struggle with categorical tuning knobs that create a hard-to-navigate optimization space.

To address these challenges, we introduce AXE, a novel learned LSM tuning paradigm that decomposes the tuning task into two steps. First, AXE trains a *learned cost model* using existing performance modeling or execution logs, acting as a surrogate cost function in the tuning process. Second, AXE efficiently generates arbitrarily many training samples for a *learned tuner* optimized to identify high-performance tunings using the learned cost model as its loss function. This task decomposition approach generalizes well for tuning simple and complex LSM designs and requires no retraining, allowing AXE to be used for tuning at scale. Compared to BO, AXE recommends higher performing tunings than BO 71% of the time while incurring 100× smaller tuning overhead. We further show that AXE requires less domain knowledge to produce optimal tunings than traditional expert-configured tuning pipelines. Lastly, we compare AXE to both state-of-the-art machine learning methods and analytical methods to show that AXE outperforms all other LSM tuning baselines.

PVLDB Reference Format:
Andy Huynh, Anwesha Saha, Harshal A. Chaudhari, and Manos Athanassoulis. AXE: A Task Decomposition Approach to Learned LSM Tuning. PVLDB, 18(13): 5582 - 5595, 2025.
doi:10.14778/3773731.3773735

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

Proceedings of the VLDB Endowment, Vol. 18, No. 13 ISSN 2150-8097.
doi:10.14778/3773731.3773735

Characteristics	Optimizer	BO-based	AXE
Ease-of-use	×	~	✓
Ease-of-develop.	×	~	✓
Tune-at-scale	~	×	✓
Sample Efficient	N/A	✓	✓
Sampling Cost	N/A	×	✓

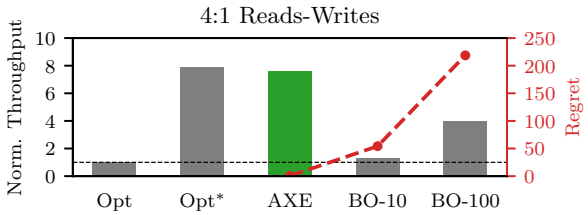


Figure 1: Tunings from AXE perform as well as a domain-expert configured optimizer (Opt*) while requiring considerably less domain knowledge to operate. Compared to Bayesian Optimization (BO) techniques for 10 and 100 iterations, AXE provides better tunings with less upfront cost.

1 INTRODUCTION

LSM Trees Are Widely Used. Due to their flexibility, efficient ingestion, and competitive reads, Log-structured merge trees (LSM trees) are a popular choice of data structure for key-value and relational data systems. For example, systems like MyRocks [41], RocksDB [18], Apache Cassandra [6], LevelDB [21], DynamoDB [17], and Apache AsterixDB [4] all utilize LSM trees. However, as the complexity of modern applications continues to grow, the requirement to correctly *tune* LSM trees becomes a harder problem.

Tuning LSM Trees. The growing number of tuning knobs is responsible for the flexibility of LSM trees, but they come at the cost of increased tuning complexity. Prior work traditionally relies either on expert tuning [45] or on a combination of analytical cost models and numerical solvers to select the appropriate value for each tuning knob [14, 24]. While these methods can provide high-performing tunings, they have several drawbacks. In particular, developing a cost model and integrating it with a numerical solver demands extensive domain knowledge; one must have a deep understanding of the system's behavior for modeling and the inner mechanics of the numerical solver to produce a valid configuration.

LSM Tuner Design Goals. Given the design complexity, the need for massive deployment, and the ever-increasing execution logs, we identify three key design goals for tuning LSM trees.

(G1) Decouple tuning effectiveness from domain expertise of the LSM internals and the optimizer used.

To acquire high-quality tunings, one requires an accurate cost model to use as the objective function in a numerical solver. However, developing an accurate cost model requires extensive domain knowledge of the system, and using it as an objective function comes

with additional constraints and an understanding of the numerical solver. For example, numerical solvers may require the objective to be continuous to produce optimal tunings, but LSM trees often have tuning knobs with categorical or discrete values, which leads to discontinuities in the cost model. Furthermore, selecting a poor initial point from which the solver navigates the cost surface can lead to low-quality solutions (details in §5). Therefore, our first design goal is to build a tuning method that does not solely rely on developing a cost model that fits an optimizer.

(G2) Tuning at Scale with Fast Time-To-Deploy.

To deploy multiple LSM tree instances across various applications – each having a unique environment and workload – requires a tuning method that can *quickly* provide *high-performing tunings* while avoiding unnecessary deploy-time penalties. For example, iterative tuning methods that leverage Bayesian Optimization (BO) can adapt to new workloads [2, 30]; however, to recommend high-performing tunings, they require multiple workload executions before deployment to receive feedback. This is particularly problematic when a workload has not been previously encountered. Therefore, our second design goal is to develop a method that adapts to unseen workloads without paying any deploy-time penalties.

(G3) Leverage past execution examples.

Lastly, every new LSM configuration deployed and workload executed is a valuable data point that can be used to improve future tuning recommendations. Therefore, our third design goal is to efficiently incorporate new examples into an offline tuning process without expensive retesting or replaying of workload traces.

AXE: A Task Composition Approach to Tuning. With these design goals in mind, we propose AXE, an LSM tuning strategy that generalizes across multiple environments and efficiently recommends high-performance configurations. Leveraging recent developments in neural networks and discrete optimization [29, 36], we decouple *cost estimation* from *cost minimization* by training different neural networks for each task. The first neural network, termed *learned cost model*, specializes in estimating the cost of executing a workload on a particular LSM configuration. The latter, termed *learned tuner*, is trained to recommend the best tuning that minimizes the cost for a specific workload. Figure 1 shows the performance of various tunings from different techniques. AXE recommends tunings with comparable performance to an expertly developed cost model optimized with an expert set-up numerical solver. Compared to iterative online BO-based methods, the offline nature of AXE requires virtually zero upfront exploration cost, thereby striking a favorable performance and usability tradeoff.

AXE Achieves Our Design Goals. Conventional tuning strategies that employ a unified learning approach require a large dataset of optimal tuning examples covering various scenarios to learn how to identify optimal tunings. This implies that any dataset we use for a single neural network approach must contain examples of optimal LSM configurations for various workloads. Compared to this approach, our learned cost model estimates the cost of executing a workload on any arbitrary tuning, optimal or not. This enables us to learn from any historical execution logs, both *eliminating the need for a domain expert* that would provide optimal tunings, and allowing us to *leverage any past example* regardless of the measured performance. AXE then uses this learned cost model as the primary way to give feedback to the learned tuner. During the offline training phase, every recommendation from the learned tuner is evaluated using the learned cost model, which bypasses any expensive database executions. This allows AXE to *effectively generalize* to environments that may *not* be represented in historical logs without resorting to online testing. Lastly, because the tuner is trained offline, AXE enables *tuning at scale* by providing high-performing LSM tunings instantaneously for any workload.

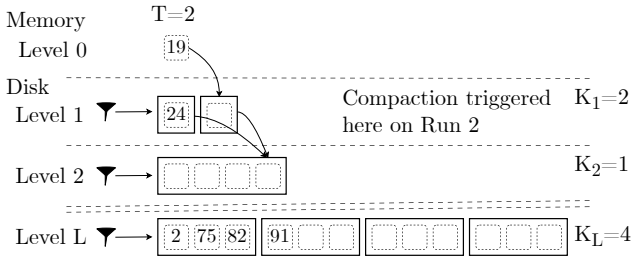


Figure 2: An LSM tree in the KLSM design space. K_i corresponds to the maximum number of sorted runs at Level i.

Contributions. Our work presents the first neural network-based task decomposition strategy for tuning LSM trees. We summarize our technical and practical contributions as follows:

- We identify the key challenges and design goals of the current state-of-the-art for automatic LSM tuning (§3).
- We introduce an offline task decomposition approach to the LSM tuning problem called AXE. Our algorithm is generalizable and easy to implement as it can learn from historical data or a predefined cost model to recommend near-optimal tunings (§4).
- We demonstrate that AXE lessens the need for domain expertise to configure a tuning pipeline that produces high-performing tunings. Compared to an out-of-the-box optimizer, AXE recommends tunings that perform up to 75% better, and compared to a domain expert configured tuning pipeline, 88% of the tunings are within 10% of the cost (§5.1 and §5.2).
- We evaluate AXE against analytical and ML-based SOTA LSM tuning approaches to show that, on average, AXE recommends tunings that outperform all other baselines. We verify these findings both analytically and with RocksDB experiments (§5.3).
- We analyze the generalizability of AXE to extend to different design spaces and demonstrate AXE scales with the number and complexity of tuning knobs. While comparing AXE to online BO-based methods, we scale down the search space in favor of BO and allow it to run 100 expensive known workload executions. Further, we show that the cost to produce a tuning from AXE is negligible. On average, to search for a tuning that performs 10% better, BO pays a 100× overhead cost for its iterative observations. We show that AXE, even if not exposed to these specific workloads, still outperforms BO 71% of the times (§5.3).

2 PRELIMINARIES AND BACKGROUND

In this section, we provide the necessary background on LSM trees and introduce the notation used for tuning.

2.1 LSM Basics

LSM trees [43] are key-value data structures that use an out-of-place ingestion paradigm. Operations modifying the tree are initially stored in memory in a write buffer of size m_{buf} holding entries of fixed size E. Once the buffer is full, it is sorted and flushed to secondary storage as an immutable *sorted run*. On secondary storage, sorted runs form logical levels growing exponentially in size, controlled by a tunable parameter *size-ratio*, denoted as T.

Compactions. We label logical levels as *Level i*, with Level 0 as the memory buffer; subsequent levels on the secondary storage are denoted as Level 1 to L. Each level holds $(T-1)$ sorted runs, exceeding which triggers an operation called a *compaction*. The two classic compaction strategies are Leveling and Tiering. In Leveling [43], incoming sorted runs from Level $i-1$ are eagerly sort-merged with Level i. Once Level i receives T sorted runs, it flushes them to Level $i+1$. Tiering lazily merges sorted runs and performs a full merge and flush to Level $i+1$ only when Level i accumulates all runs. Only the most recent valid entry is kept during these compactions; any older entries are overwritten with the newer data.

Hybrid Compaction Policies. Between Leveling and Tiering, there is a continuum of hybrid compaction policies that expose tuning knobs to navigate the design space. We use the KLSM model to describe these hybrid policies, where each Level i has a sorted run capacity, denoted by K_i, that ranges from 1 to $T-1$ [24, 42]. The total amount of data per level remains the same as in classic strategies, but the sorted run sizes change depending on the level capacity. Thus, we describe Leveling ($K_i = 1$ for all i), Tiering ($K_i = T-1$ for all i), and hybrid compaction policies such as the FluidLSM, where the first $L-1$ levels have the same capacity $K_i = K_1$ for $i \leq L-1$, and the last level is assigned a different capacity [16].

Operations. LSM trees support three basic operation types: point queries, insertions, and range queries. A *Point Query* starts at the buffer and continues through each subsequent level, from Level 1 to Level L. At each level, the query begins with the most recent sorted run and ends with the oldest. If the query finds a matching key, it terminates and returns the result. *Insertions* are immediately appended to the in-memory write buffer. Every insertion includes either a key-value pair to indicate a write, an existing key to signify an update, or a unique value to denote a deletion. *Range queries* first scan the buffer and then traverse the remaining sorted runs on disk to find potential matches.

Optimizing Reads. Traditional LSM trees assign the same false positive rates to Bloom filters at every level, leading to inefficient memory usage and redundant disk accesses during read operations. Therefore, we use the schema in Monkey [14] where each Bloom Filter is optimally allocated the correct amount of memory by setting the false positive rates proportional to run sizes.

High Impact Tuning Knobs. LSM trees expose multiple tuning knobs that must be configured appropriately by the user for optimal performance. Prior work demonstrates an often non-linear relationship between these knobs and system performance [14, 24, 39, 47, 59]. From these works, we identify size ratio (T), balancing memory allocation between Bloom Filters (m_{filt}) and the write buffer (m_{buff}), and choice of compaction policy as significant contributors to LSM performance. For compaction policy, we either use a classic strategy (Tiering or Leveling), the FluidLSM design space, or the KLSM design space. If we select π = KLSM, we additionally must decide on values for K_i. Similarly, for π = FluidLSM, we must select the capacity for the first $L-1$ levels and the last level. Putting everything together, we define an LSM tuning as follows.

Table 1: Summary of cost model notation.

Term	Definition
E	Size of a key-value entry
B	Number of entries that fit in a page
N	Total number of entries
H	Total memory budget
m_{buff}	Memory allocated for the write buffer
m_{filt}	Memory allocated for the Bloom Filters
T	Size ratio between consecutive levels
$L(T)$	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
$f_i(T)$	Bloom filter false positive rate at level i with size ratio T
f_a	Read/write asymmetry ratio for storage device
f_{seq}	Cost of a sequential read w.r.t. a random read
S_{RQ}	Range query selectivity

Definition 1 (Tuning). *Let φ be a set of values associated with the identified high-impact tuning knobs $(m_{\text{filt}}, m_{\text{buff}}, T, \pi)$, where π is the choice of compaction policy.*

2.2 Cost Model

We use the state-of-the-art cost model for LSM trees, as defined in prior work, which is shown to accurately capture the estimated I/Os per query as we vary tunings and workloads [24].

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 from the Bloom filters. Therefore, the expected number of I/Os per level depends on the Bloom filter memory allocation at that level, where each level has at most K_i runs with equal false positive rates.

Non-empty Point Query Cost (Z_1). The non-empty point query cost consists of two cost components: first, the cost of finding the result at Level i, which is proportional to the size of the level. This is calculated by $\frac{(T-1)\cdot T^{i-1}}{N_f(T)} \cdot \frac{m_{buf}}{E}$, where $N_f(T)$ represents the number of entries in a full tree up to $L(T)$ levels. Secondly, it depends on the chance of a failed I/O at preceding levels due to false positive results of Bloom Filter given by $\sum_{j=1}^{i-1} f_j(T)$ and additional I/Os to account for the entry being found in the middle of the run on average given by $\frac{(K_i-1)}{2} \cdot f_i(T)$.

Range Query Cost (Q). Range queries in LSM trees involve merging valid entries across multiple sorted runs per level. Each leg of the range query performs a multi-way merge between the runs to produce a single sorted output. At each level, there are at most K_i disk seeks that correspond to the capacity of sorted runs. Once the initial disk seeks are completed, the remaining accesses are sequential. The cumulative number of pages scanned is given by $S_{RQ} \cdot \frac{N}{B}$, where S_{RQ} is the selectivity of the range query. After the initial seek, sequential I/O scans are used for subsequent pages and

Table 2: Summary of the key cost functions for Log-Structured Merge (LSM) tree operations. [24] These cost functions estimate the average number of I/O associated with each operation type.

Operation	Formula	Description
Empty Point Query	(1) $Z_0(\varphi) = \sum_{i=1}^{L(T)} K_i \cdot f_i(T)$	Dependent on false positive rates of Bloom Filters across levels
Non-empty Point Query	(2) $Z_1(\varphi) = \sum_{i=1}^{L(T)} \frac{(T-1)\cdot T^{i-1}}{N_f(T)} \cdot \frac{m_{buf}}{E} \left(1 + \sum_{j=1}^{i-1} K_j \cdot f_j(T) + \frac{K_i - 1}{2} \cdot f_i(T)\right)$	Considers entry location probability and any preceding levels' false positive rates
Range Query	(3) $Q(\varphi) = f_{seq} \cdot S_{RQ} \cdot \frac{N}{B} + \sum_{i=1}^{L(T)} K_i$	Estimated by selectivity plus the total per-level disk seeks
Write	(4) $U(\varphi) = f_{seq} \cdot \frac{1+f_a}{B} \cdot \sum_{i=1}^{L(T)} \frac{T-1+K_i}{2K_i}$	Estimated using per-level merge operations, considering read/write asymmetry

are scaled by f_{seq} to reflect the cost difference between sequential and random I/Os.

Write Cost (U). We model writes under a worst-case scenario in which incoming entries propagate through all levels of the LSM tree without overlapping. We estimate the total write cost by calculating the average number of merge operations a write participates in at each level. At a single level i, a flush from the previous level $i-1$ participates in an average of $\frac{T-1+K_i}{2K_i}$ merges and accounts for both eager merges and full-level compactions. The total write cost is calculated by summing the average merge costs across levels and dividing by *number of entries per page B*, adjusted for read/write asymmetry f_a and the sequential I/O cost f_{seq}.

Calculating Cost. To calculate the total cost, we represent a workload as the distribution over operation types.

Definition 2 (Workload). *Let $w = (z_0, z_1, q, u)$, where z_0, z_1, q, and u are the percentage of empty point queries, non-empty point queries, range queries, and writes, respectively.*

Additionally, we define a set of system parameters

Definition 3 (System). *Let $S = (E, B, H, N, f_a, f_{\text{seq}})$ be a set of system parameters and $\mathcal{S}$ be the set of all possible systems.*

Note that Table 1 contains the description of each variable. Then, the total cost parameterized by a set of system parameters is simply the weighted sum across each operation or

$$C_S(w, \varphi) = z_0 \cdot Z_0(\varphi) + z_1 \cdot Z_1(\varphi) + q \cdot Q(\varphi) + u \cdot U(\varphi). \quad (5)$$

We also refer to each component of the cost function by its respective operation subscripts; for example, for empty reads, the cost is $C_{z_0} = z_0 \cdot Z_0(\varphi)$.

3 PROBLEM DEFINITION

In this section, we formalize the LSM tuning problem and describe its associated challenges.

3.1 The Tuning Problem

When managing LSM tree instances, database administrators are tasked with determining the optimal configuration for a specific workload to minimize costs (e.g., latency or cloud costs). We denote the problem as the Tuning Problem, which we formally define as

Problem 1 (Tuning Problem). *Let Φ be the set of all potential LSM configurations. Given a specific workload $w \in \mathcal{W}$, and a cost $C : W \times \Phi \to \mathbb{R}$, find the LSM tree tuning φ^* such that*

$$\varphi^* = \arg\min_{\varphi \in \Phi} C_S(w, \varphi) \quad (6)$$

When solving Problem 1, we denote the minimum cost by C^*.

3.2 Challenges

Given the complex design space of LSM trees, there are various challenges in solving Problem 1 for *all possible workloads*. We focus on some significant challenges in the offline setting, specifically, the calculation of cost $C(w, \varphi)$, efficiently navigating the tuning knobs space, and alleviating the need for expert domain knowledge.

Calculating Cost. When calculating the cost of a workload executed on a configuration, $C_S(w, \varphi)$, we often have two choices: either run the workload on a deployed database with the respective configuration and wait for the result, or proxy the execution by using a model to estimate the cost. In the first case, executing on a physical database is expensive, as we must wait for the total execution time to observe the cost. Additionally, if we change a tuning knob that changes the internal layout of data, we are required to rebuild the database after each change. For example, to calculate the cost between two configurations with differing size ratios, we must build both LSM instances before executing a workload so that the underlying files respect each size ratio. If all tested configurations are suitable, this may not be too prohibitive; however, if we are required to deploy a slow configuration during the tuning process, we pay a significant penalty waiting for each execution to finish.

Instead, an analytical cost model allows for constant time estimates of the cost. The key challenge is creating the cost model, which requires a designer to have *intimate knowledge of the database system mechanics* [28, 51]. This leads to an excess of man-hours spent modeling the system before we can use the cost model to tune it, and this process must be repeated if it introduces new features or tuning knobs. Additionally, cost models may be inaccurate because they are estimates. This can partially be attributed to the difficulty of capturing the intricate relationships among sets of tuning knobs. The inherent approximations in cost models may translate to suboptimal or even problematic configuration recommendations.

Tuning Knobs with Categorical Values. As some knobs take on categorical values, the LSM tuning problem is classified as a

mixed integer optimization problem, which is NP-hard [8]. Therefore, when using an optimizer, designers typically approximate categorical values using a continuous space to solve an easier constrained optimization problem [14, 16, 23, 24, 34]. The solution is then rounded off to create a feasible tuning; however, this introduces errors in the optimization process and can trap the optimizer in local minima, leading to suboptimal tunings. This error only increases as the number of categorical decision variables increases [46].

Requiring Dual Domain Expertise. To work around the challenges of cost calculation and categorical tuning knobs, a designer must have a deep understanding of both the system and the optimizer. Understanding the system is necessary to create the cost model, which must also satisfy any requirements imposed by the optimizer. For example, optimizers often only produce feasible solutions if the objective function is continuous, however, categorical tuning knobs inevitably introduce discontinuities when modeling system behavior. Therefore, the designer must understand the underlying algorithms used in the optimizer to adjust the cost model appropriately to produce optimal tunings.

3.3 Defining a Strategy

Note that the Tuning Problem is expressed as a problem for a *single* database instance executing *one* specific workload; however, database administrators are often responsible for multiple different database instances serving a variety of applications. Therefore, it would be advantageous to have a reusable strategy that recommends optimal configurations for all possible workloads. We refer to this strategy as a *tuner*, which we define as follows:

Definition 4 (Tuner). *A tuner is a function* $g : W \times \mathcal{S} \rightarrow \Phi$ *that recommends a configuration based on a given environment or workload-system pair.*

Designers often use different strategies to create a feasible tuner that recommends near-optimal tunings. For example, with access to performance logs of historical workloads across various database configurations, we may choose to train a single tuner neural network using these historical traces. However, to train a network that recommends near-optimal tunings, we require examples with optimal configurations for each workload. In such a case, it is infeasible and expensive to create a sufficiently large dataset capable of training a network by relying on domain experts. Another common technique is creating a feedback loop using Bayesian Optimization (BO), where a model suggests a tuning to deploy, and iteratively refines the recommendation based on observed performance metrics. However, such an iterative method adds additional upfront costs to explore the design space. We have explained how building a tuner is particularly challenging; next, we discuss our approach.

4 AXE: A LEARNED TUNER

In this section, we introduce our approach and the architecture of AXE, a dual neural network solution for creating a tuner that recommends near-optimal tunings for any workload.

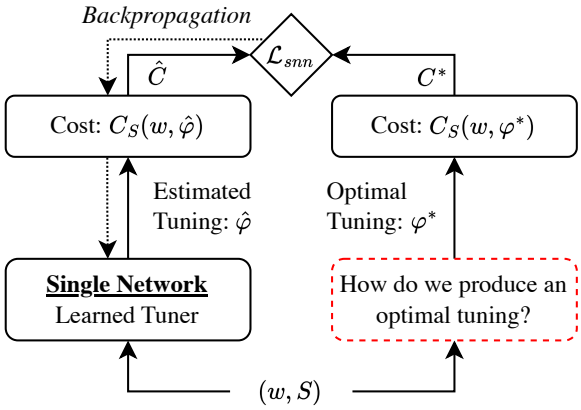


Figure 3: Training loop for a single neural network approach. This approach requires a differentiable cost function and a training dataset consisting only of optimal tunings.

4.1 The Learning Approach

Challenges Training a Single Neural Network. Suppose we wish to train a single neural network to take as input a workload-system (w, S) pair and output a tuning φ. In a supervised learning setting, our loss function calculates the difference between the cost of our network's recommended tuning and the optimal tuning. Let $g(w, S; \theta_t)$ be the function representing a neural network where θ_t denotes the weights and biases of our learned tuner and $\mathcal{D}$ be the dataset with tuples of workloads, system parameters, tunings, and their associated cost, (w, S, φ, C). Then, the tuner loss function is

$$\mathcal{L}^*(\theta_t) = \mathop{\mathbb{E}}_{(w,S)\sim\mathcal{D}} \left[||C_S(w, g(w, S; \theta_t)) - C^*||_p \right] \tag{7}$$

where we can select any L_p norm. Although this loss formulation is common practice in supervised learning, several issues make it infeasible for our problem.

Obtaining the Correct Data: Firstly, dataset $\mathcal{D}$ is not guaranteed to contain optimal tunings for each workload; therefore, we must calculate C^* for each example. Secondly, calculating C^* is an optimization version of the decision problem – Problem 1 (finding a min vs. arg min); they are equivalent problems. This implies we must solve Problem 1 for each unique (w, S) pair to create an appropriate dataset – an infeasible task. Figure 3 shows a block diagram of the necessary training framework and highlights the issue of requiring us to find C^* to train a single neural network.

Cost Constraints: Since we wish to train the neural network using the backpropagation algorithm, the cost function must be differentiable and continuous [7, 20] – a challenging task when modeling discrete decision variables in the cost function. For a neural network to output a discrete value, such as size ratio, we must use a softmax output layer [32]. The softmax layer outputs a probability distribution, with the highest probability corresponding to the predicted size ratio; however, to compute our tuning cost with this size ratio, we must convert the probability distribution to either a one-hot encoded representation or a discrete numerical value. This conversion can only be done using arg max, an operator that is *not* differentiable [29, 40]. As many of the standard LSM tuning knobs,

such as size ratio, take on discrete values, this further illustrates the difficulties of a single neural network approach.

Avoiding Calculation of C^*. Instead, we look at an unsupervised learning setting where our loss function is simply the cost of executing the recommended tuning by our neural network,

$$\mathcal{L}^*(\theta_t) = \mathop{\mathbb{E}}_{(w,S)\sim\mathcal{D}} \left[C_S(w, g(w, S; \theta_t))\right] \tag{8}$$

Because we minimize this loss during the training process, the resulting neural network is designed to recommend the optimal configuration for any given (w, S), effectively solving Problem 1. While this alleviates the issue of our dataset needing examples of optimal configurations, we still face the same fundamental problems with this approach. Categorical values represented by one-hot encoded vectors still require special consideration to satisfy the differentiability requirement without using the arg max operator, and we still need a differentiable cost calculation.

Splitting the Problem. To solve the above issues, we decompose the problem into two tasks: (i) finding a differentiable and continuous surrogate cost function and (ii) using a gradient-based method to optimize the surrogate function. Figure 4 shows the overall architecture of our dual neural network approach. The first neural network acts as our surrogate for a differentiable and continuous cost function that accepts discrete encoded representations of categorical values as inputs. This network allows for constant-time estimations of the performance of any LSM tuning on a workload. The second network assumes the role of a tuner that recommends cost-minimizing configurations across the space of heterogeneous tuning knobs.

Finding a Surrogate Cost. We use a neural network that takes the same inputs and outputs the same values as the cost model. Let $f(w, S, \varphi; \theta_c)$ be the function representing a neural network, where θ_c denotes the weights and biases of the learned cost model. The loss function is the difference between the estimated cost of a workload-configuration-system tuple and the actual cost,

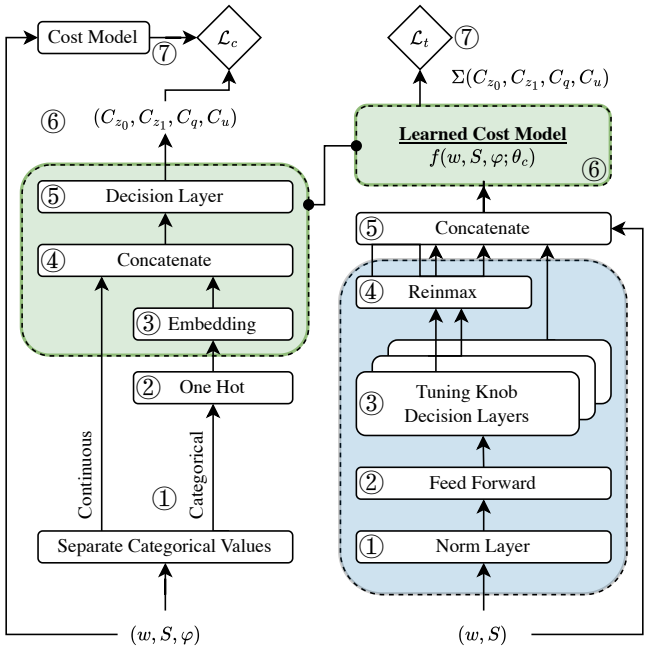
$$\mathcal{L}_c(\theta_c) = \mathop{\mathbb{E}}_{(w,S,\varphi,C)\sim\mathcal{D}} \left[||C - f(w, \varphi, S; \theta_c)||_p\right] \tag{9}$$

Figure 4a shows the training framework for the learned cost model. We refer to this surrogate function as the *learned cost model* for the remainder of this paper. The cost calculation in Equation (5) contains four components: the non-empty read, empty read, range read, and write cost. For our implementation, the learned cost model uses a multi-headed architecture to estimate the cost of each component separately. Therefore, we modify the loss function to be the summation of the differences of each component. Training a model with this loss yields a learned cost model, f, that can replace the cost calculation, C, for the learned tuner, while being differentiable. This addresses issues highlighted in the single neural network approach, allowing us to utilize gradient methods to find an optimal tuning.

Rewriting Tuner Loss. With a differentiable and continuous surrogate cost function f, we replace the cost calculation in Equation 8 to formulate a feasible loss function,

$$\mathcal{L}_t(\theta_t) = \mathop{\mathbb{E}}_{(w,S)\sim\mathcal{D}} \left[f(w, S, g(w, S; \theta_t); \theta_c)\right] \tag{10}$$

Note that we sample (w, S) from the same dataset $\mathcal{D}$ as the loss calculation only requires the workload and system. Figure 4b shows



(a) Learned Cost Model **(b) Learned Tuner**

Figure 4: The model architectures for (a) the learned cost model and (b) the learned tuner. We pass the differentiable operators of the learned cost model to the learned tuner to calculate tuning performance.

the training framework. The learned tuner recommends a tuning that is passed to the learned cost model f alongside the original workload-system tuple to calculate the cost to be minimized. We can train a tuner using this loss function, as the learned cost model neural network represents a continuous and differentiable universal approximator of an unknown nonlinear cost function. We refer to this model as a *learned tuner* for the remainder of this paper.

4.2 Learned Cost Model

Algorithm 1: Training Learned Cost Model

Input: Learned cost model architecture f, dataset $\mathcal{D}_{\text{Cost}}$, learning rate η_c, number of epochs N_e
Output: Learned cost model weights and biases θ_c

```
Randomly initialize all weights and biases θ_c
for e = 0 to N_e
    Partition D_Cost into batches, denoted by β
    for each b ∈ β
        L_c ← (1/|b|) · Σ_{(w,S,φ,C)∈b} (C − f(w, S, φ; θ_c))^2
        θ_c ← θ_c − η_c · ∇_{θ_c} L_c
return θ_c
```

Data and Training. To train the learned cost model, we use a dataset with tuples of workloads, system parameters, LSM tunings, and their respective performance $(w, S, \varphi, C(w, \varphi))$ which we refer to as $\mathcal{D}_{\text{Cost}}$. These tuples may either be sourced from real-world

historical traces or generated from a user-defined analytical cost model that may not meet the criteria of being differentiable and continuous. For example, a cloud vendor that offers databases as a service may quickly obtain the dataset by monitoring their deployed instances. However, we do not have access to historical traces from large-scale deployed databases. Creating the dataset from scratch presents challenges, as most knobs will alter the underlying data layout in LSM-based data systems. For example, when testing two LSM configurations with differing size ratios T_1 and T_2, a snapshot of the database for T_1 could never be used to test the other, as the structure of the tree would be invalid w.r.t. size ratio T_2. Instead, we must rebuild the database with the size ratio set to T_2 before testing any workload. In most instances, the cost of rebuilding the database is significantly greater than executing the workload.

Therefore, we generate w_{Cost} with random workloads, system parameters, and tuning knob values, then leverage the analytical cost model defined in Section 2 to compute cost as it accurately captures the average I/Os per query for LSM trees. Workloads are sampled uniformly at random, while the system and tuning knob values are sampled within a predefined range of common parameters. For example, page sizes range from 2KB–16KB as they are standard for most systems. The training follows Algorithm 1, where each epoch is a pass over the entire dataset. Additionally, we leverage techniques such as minibatch gradient descent to train over a dataset that does not fit in memory [20].

Neural Network Architecture. Figure 4a shows the overall architecture of the learned cost model. ① We first separate the categorical variables, such as size ratio, from the continuous variables. ② All categorical values are one hot encoded and passed through ③ an embedding layer. Note that we use a dedicated embedding layer for each individual categorical tuning knob. For example, in a KLSM design space, the model has a different embedding layer for each value of K_i, corresponding to each level i, alongside a separate embedding for size ratio T. ④ We then concatenate the outputs of the embedding layers with the continuous variables from the original input. We pass the now embedded *feature vector* through ⑤ a decision layer, which divides the output ⑥ into four output *heads*, each representing the estimated cost of a component. ⑦ Finally, the loss is calculated as the difference between the estimated cost components and the cost components calculated from the analytical cost model defined in Section 2, which allows us to backpropagate gradients of the loss to update weights and biases.

4.3 Learned Tuner

Data and Training. The learned tuner takes as input a workload and additional system parameters and outputs a tuning for which we evaluate the performance with the previously trained learned cost model. Therefore, our dataset, $\mathcal{D}_{\text{Tuner}}$, is composed of tuples of generated workloads and system parameters (w, S). Note that these workload-system parameter pairs are not necessarily the same ones seen in $\mathcal{D}_{\text{Cost}}$; instead, we can augment the dataset by artificially generating a large number of workload-system pairs. Because the dataset does not rely on calculating a cost, we can scale its size up or down depending on the desired training budget (i.e., compute or time constraint). For training, we follow Algorithm 2 where we provide a *pre-trained* learned cost model to calculate the loss $\mathcal{L}_t$.

Algorithm 2: Training Learned Tuner

Input: Learned tuner architecture g, learned cost model f and its weights θ_c, dataset $\mathcal{D}_{\text{Tuner}}$, learning rate η_t, number of epochs N_e
Output: Weights and biases θ_t

Randomly initialize all weights and biases θ_t
for $e = 0$ *to* N_e
  Partition $\mathcal{D}_{\text{Tuner}}$ into batches, denoted by β
  for *each* $b \in \beta$
    $\mathcal{L}_t \leftarrow \frac{1}{|b|} \cdot \sum_{(w,S)\in b} f(w, S, g(w, S; \theta_t); \theta_c)$
    $\theta_t \leftarrow \theta_t - \eta_t \cdot \nabla_{\theta_t} \mathcal{L}_t$
return θ_t

Architecture. Figure 4b shows the architecture of the learned tuner portion of AXE during training. The input is first passed through a ① normalization layer to account for different scaling of values. Next, ② the output of the feed-forward network is redirected to ③, a series of decision layers with outputs corresponding to each tuning knob. The decision layers that correspond to a categorical tuning knob output the logits vector that is passed through a ④ Reinmax operator to encode a discrete valued output [36]. This allows the network to represent categorical tuning knobs as discrete values while still enabling optimization via the backpropagation algorithm, as the Reinmax operator is differentiable. Using the output decisions, the tuning knob settings are ⑤ concatenated and passed on to ⑥ the learned cost model to be evaluated. The ⑦ feedback serves as the loss for our learned tuner to know which tuning knobs decrease the estimated total cost[1].

4.4 Why Split the Tuning Process

By splitting the tuning process, we replace a complex problem with two simpler problems, the benefits of which we discuss below.

① Decoupling Tuning Prediction from Cost Estimation. By specializing each neural network to a specific task – cost prediction and tuning optimization – each network requires fewer learned parameters (weights and biases) and fewer samples to train effectively. The learned cost model only models the relationship between the inputs (w, S, φ) and the cost C and is not concerned with finding an optimal tuning. On the other hand, the learned tuner does not need to estimate an accurate cost; instead, it recommends the tuning knob values that lead to a lower predicted cost.

② Generalizability. Similarly, while we require fewer samples to train effectively, our method allows us to train a tuner on a broader range of scenarios. To train a single neural network, the training data must contain workload-system pairs with a corresponding optimal cost C^*; this implies we must solve the tuning problem before training a single neural network, as discussed in § 4.1. In contrast, our method uses *any* data points from historical executions, including instances with poor performance, as it enables the learned cost model to learn low-performance neighborhoods of the cost surface.

[1]Note that the one-hot operator in the learned cost model ⑥ is non-differentiable. Therefore, we bypass the one-hot operator from Figure 4 while evaluating a tuning from the learned tuner to facilitate training using the backpropagation algorithm.

③ **Data Augmentation.** In AXE, we create a guided feedback loop between the cost model and the tuner, where the cost model evaluates the learned tuner's predicted tuning. With the learned cost model, we can explore regions of the parameter space with lower costs that may not be represented in the original training data. Therefore, the learned tuner's dataset can be augmented with generated examples of workload-system pairs that are not necessarily a part of the original training data. This approach provides faster convergence to a better performance.

④ **Mitigating Error Propagation.** In a single neural network approach, potential errors in the cost estimation directly affect the ability to generate optimal tunings. In contrast, task decomposition allows a designer to mitigate potential errors at two levels (learned tuner and learned cost model). For example, a learned cost model does not need to accurately estimate costs in low-performance neighborhoods of the cost surface; instead, it can focus on only accurately modeling high-performance neighborhoods. Similarly, a learned tuner does not need to accurately model tunings that correspond to high estimated costs, reducing its decision space. Errors in the cost estimation may still affect the downstream tuning output; however, we can fine-tune each model individually to mitigate its errors, leading to more robust performance in the overall task.

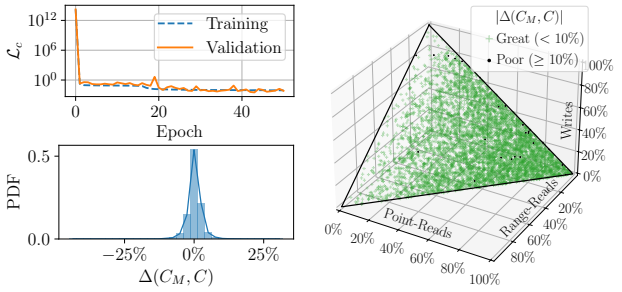
⑤ **Iterative Development.** The separation of tasks allows us to iteratively improve both models independently of one another as additional observations from a deployed database become available, including those with low performance. This iterative approach to performance improvements leads to more efficient use of newly available training data to fine-tune existing models, unlike a single neural network-based approach that can only utilize optimal performance observations and requires end-to-end retraining.

Thus, the task decomposition of the tuning problem into multiple learning problems achieves our original design goals. It allows us to make more targeted improvements, *(G1)* eliminates the need for a domain expert by learning from past executions, *(G2)* enables us to tune at scale as our neural networks provide tunings in constant time at deployment, and *(G3)* can effectively use new training data from a dynamic environment where databases are deployed.

5 EVALUATION

In this section, we analyze AXE's ability to learn the database's cost surface and evaluate the quality of tunings under various scenarios, showing that AXE performs as well as a domain-expert-configured tuning pipeline. We demonstrate that our Learned Cost Models can accurately capture LSM performance (§5.1), and the Learned Tuner can match the quality of an expertly-tuned analytical solver with an accurate cost model (§5.2). We then show that AXE can successfully replace and often improve upon SOTA LSM tuning methods (§5.3), by comparing three approaches: an offline analytical one (Monkey [14]), an online ML-based approach (CAMAL [54]), and an online Bayesian Optimization-based baseline inspired by recent research [3]. Lastly, we analyze the generalizability of AXE and present evidence that it scales with the number of tuning knobs, compared to other data-driven methods.

Methodology. Our server is equipped with a 13th Gen Intel Core i9-13900KF processor, 128 GB of RAM, and a 20 GB NVIDIA RTX 4000 Ada Generation graphics card, running Ubuntu 22.04.4 LTS. We use



(a) Training loss and performance over test set

(b) Performance distribution over workload $|\Delta(\cdot)| \leq 10\%$

Figure 5: The performance of the learned cost model of AXE. The probability density shows 99% of cost estimates are within a 10% band of normalized cost difference. Points outside this 10% band are sparsely distributed across the space of potential workloads.

Python 3.11 with PyTorch version 2.1. We use the Machine Learning Optimization Service (MLOS) library version 0.6.0 by Microsoft as the BO baseline implementation, as it is primarily used to tune relational database management systems [13, 19, 30, 31]. We utilize a fork of RocksDB [18] where we implement the KLSM compaction strategy [24, 35] and allocate bloom filter memory based on the Monkey schema [14, 15] as it improves performance.

5.1 Learned Cost Model Performance

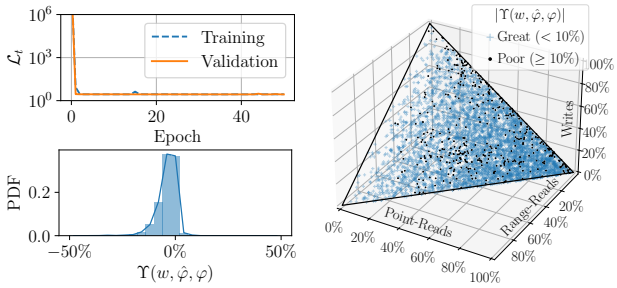
Normalized Cost Difference. To evaluate the learned cost model's performance, we measure the normalized difference in the estimated cost to the true cost as,

$$\Delta(C_M, C) = \frac{C - C_M}{C} \tag{11}$$

where C_M is the cost estimated by the learned cost model. We express the normalized cost difference as a percentage.

Training and Setup. We use a dataset with $|\mathcal{D}_{Cost}| = 10^6$ tuples to train the cost model. This dataset is split such that 90% of the data is used for training and 10% is used for validation. For the loss function, we use the mean-squared error – square of the L_2 norm of Equation 9. We use embedding layers with an output size of 8, and the decision block is a feed-forward network with 3 hidden layers, each with a width of 64. This results in a learned cost model with 29,000 parameters, or 0.12MB of space. We use the Adam optimizer with a starting learning rate of 10^{-3}. For faster convergence, we adjust the learning rate at each epoch according to the cosine annealing schedule, as implemented in PyTorch [37]. We reserve 5000 unseen examples $(w, S, \varphi, C) \notin \mathcal{D}_{\text{Cost}}$ for testing purposes. For brevity, we only present results for the most complex and challenging design space, namely KLSM.

Using Learned Cost Model as a Surrogate. Figure 5a shows the magnitude of loss through the training duration and the histogram of normalized cost difference over the test suite of unseen examples. We observe that 99% of the cost estimates from the learned cost model are within a 10% normalized cost difference $(-10\% \leq \Delta(C_M, C) \leq 10\%)$. Figure 5b visualizes this performance



(a) Training loss and test suite for the learned tuner

(b) Performance distribution over workload $|\Upsilon(\cdot)| \leq 10\%$

Figure 6: The performance of the learned tuner in AXE. Over 88% of the tunings provided by AXE are within a 10% performance band from our domain expert configured optimizer.

band over potential workloads[2]. We observe that the learned cost model performs well across all contexts, with a minority of examples outside the 10% performance band.

5.2 Learned Tuner Performance

Normalized Performance Difference. We compare two tunings φ_1 and φ_2 using the normalized difference between their performance while executing a single workload as follows,

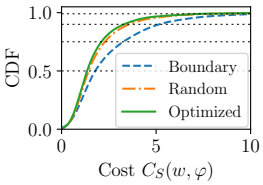
$$\Upsilon(w, \varphi_1, \varphi_2) = \frac{C_S(w, \varphi_2) - C_S(w, \varphi_1)}{C_S(w, \varphi_2)}. \quad (12)$$

Values of $\Upsilon(w, \varphi_1, \varphi_2) > 0$ implies φ_1 performs better than φ_2.

Training and Setup. The learned tuner only requires a workload w and a system S as inputs. Hence, we augment the dataset $\mathcal{D}_{\text{Cost}}$ to create a larger $|\mathcal{D}_{\text{Tuner}}|$ of size 10^7 using additional examples obtained by sampling tuples of (w, S) at random. Similar to the learned cost model, the dataset is split such that 90% is used for training and 10% is used for validation. The neural network consists of a feed-forward block with two hidden layers, each of size 64 neurons with a ReLU activation function. Each tuning knob decision layer is a fully connected layer. This results in a learned tuner with 65,268 parameters, a size of 0.26MB. We use the Adam optimizer with an initial learning rate of 10^{-3} with the cosine annealing scheduler. We set aside 5000 unseen examples $(w, S) \notin \mathcal{D}_{\text{Tuner}}$ for testing purposes. For brevity, we present results only for KLSM; however, our findings generalize to other design spaces.

Creating an Expensive Baseline. To evaluate the performance of AXE tunings against a benchmark, we manually configure an optimizer in conjunction with the cost function to obtain high-quality tunings for each of the 5000 testing examples, referred to as the *domain expert configured optimizer* (DECO). This process requires significant manual configuration and tweaking; therefore, it is not scalable to larger datasets. We use the Sequential Least Squares Programming (SLSQP) algorithm in SciPy as an optimizer and approximate categorical tuning knobs by using continuous values and rounding them off to obtain a valid configuration. Additionally, we leverage domain expertise to manually configure the optimizer's

[2]For ease of three-dimensional visualization, we combine empty and non-empty reads on the "Point-Reads" axis. This causes the region of $z_0+z_1 = 0$ to be sparsely populated.

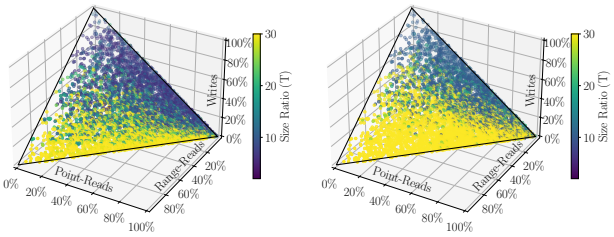


(a) Optimizer performance

(b) Median slowdown from optimal

Percentile	Boundary	Random
P50%	1.0×	1.0×
P75%	1.7×	1.1×
P90%	2.4×	1.2×
P99%	4.1×	1.7×
P99.9%	6.2×	3.3×

Figure 7: When initial values are set to the domain's boundary, we obtain the poorest-performing designs, which perform up to 6.2× worse than the optimal suggestions. While a random initialization improves performance, a domain expert must configure the optimizer for optimal performance.



(a) Domain expert configured

(b) AXE: Learned Tuner

Figure 8: Size ratio (T) decision for various workloads. Both the learned tuner and the domain expert configured optimizer select larger values of T when writes are less dominant.

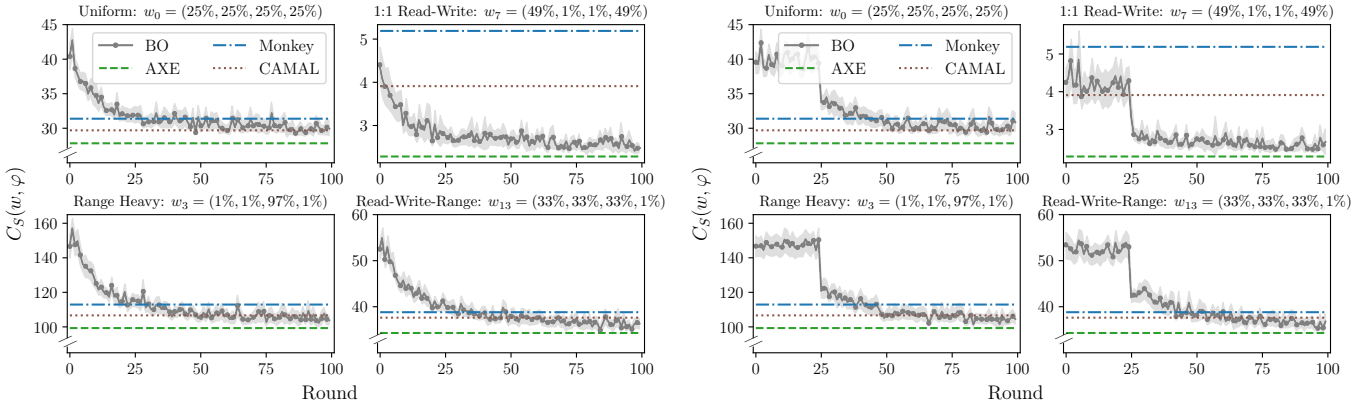
initial starting points to be close to high-performance tunings. Figure 7a plots the cumulative distribution of the cost of recommended designs for optimizers initialized with a starting point on the boundary of the domain, a random point, and an optimized starting point. We observe that the DECO performance is highly sensitive to such hyperparameters. Without optimized starting points, the solver often gets trapped in local minima, resulting in tunings that perform up to 6.2× slower. Furthermore, the starting conditions also affect the solver's time-to-solution, ranging from 1ms to several minutes.

AXE Performs On Par with a Domain Expert. When compared to the DECO tunings, 88% of tunings recommended by AXE are within a 10% performance band $(\Upsilon(w, g(\cdot), \varphi) \leq \pm 10\%)$ and 98% are within 20% as shown in the histogram of Figure 6a. We observe that for specific instances of the tuning problem, AXE recommends tunings that outperform the SLSQP optimizer. As KLSM contains numerous categorical tuning knobs (each K_i), approximating discrete values with continuous values results in the DECO suggesting suboptimal tunings. We analyze the quality of AXE's selection of size ratio in Figure 8, where we qualitatively verify that AXE recommends correct size ratios across workloads. For example, as the proportion of point-reads in a workload increases, both tuners recommend lowering the size ratio.

5.3 End-to-end Evaluation of Tuning with AXE

We now show that AXE matches or outperforms SOTA LSM tuning.

Baselines. We compare AXE to three LSM tuning techniques: (1) Monkey [15] models the worst-case number of I/Os for each potential operation type while considering the classic compaction



(a) At every round, BO suggests its best-estimated tuning.

(b) BO samples 25 random points before suggesting the best tuning.

Figure 9: AXE recommends configurations that outperform all other baselines. For the tunings provided by BO after 100 rounds of optimization for the KLSM design space with a given workload, AXE performs better than 92% of them. Note that the suggestions from AXE, CAMAL, and Monkey in each graph are produced from a single model, while we must run a new instance of BO for every workload. When we seed BO with 25 random samples, AXE still suggests better-performing tunings.

Table 3: Testing Suite of Representative Workloads

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

policies, *Leveling* and *Tiering*. Monkey is an offline method that uses the *Newton-Raphson* method to navigate its cost model. (2) CAMAL [54] uses an *active learning* paradigm that combines a cost model, polynomial regression, and tree ensembles to prune the search space of designs and learn the optimal tuning in an online fashion, and only considers the classic LSM compaction policies of Leveling and Tiering. (3) We develop a BO-based baseline by adapting recent DBMS BO-based tuning approaches [2, 3]. Such BO-based approaches are online methods for optimizing black-box functions that are otherwise expensive to evaluate. The basic optimization loop is as follows: at the start of every iteration, BO suggests a new LSM tuning to evaluate. Once the cost is calculated or observed, the value is returned to the optimizer to improve future suggestions. This process continues until a user-defined number of rounds or resource budget (e.g., machine time) is met.

Workloads & Methodology. We test all tuning methods using a set of workloads shown in Table 3, following prior work [23, 24, 54]. Each workload is categorized based on the dominant operation types to observe trends in how tunings perform for each workload type. We run the BO tuning algorithm for 100 rounds on each workload and repeat the same instance 100 times to measure BO's stochastic behavior. Note that we allow BO to run the same workload multiple times to find the optimal tuning, while AXE is not optimized for that specific workload, further favoring BO. Additionally, we run these experiments across LSM design spaces of increasing complexity, when applicable – the simplest being Tiering or Leveling [14], followed by the FluidLSM design space [16], and the complex design space of KLSM [24], totaling over 7 million evaluations. Because CAMAL and Monkey only consider the Tiering and Leveling design space, we use their recommended classic tuning when comparing performance.

AXE Outperforms All Baselines. Figure 9a shows the quality of the tuning (cost of running each workload) recommended by CAMAL, Monkey, AXE, and BO at each round. The line corresponding to BO shows the average tuning performance over 100 different instances, with the band denoting the 95% confidence interval. AXE recommends a tuning that performs better than all other baselines for each workload, an observation that extends to all fifteen workloads in Table 3. We observe that for read-intensive workloads – those dominated by range scans or point queries – all zero-overhead tuning methods produce competitive tunings. However, when writes are introduced in workload w_7, Figure 9a shows that AXE better navigates the LSM design space to recommend a tuning that performs 2× better than CAMAL and 2.5× better than Monkey. Compared to a BO-based method, AXE provides tunings that perform better 92% of the time. When using BO, some initial random evaluations are expected to seed the initial GP model. In Figure 9b, we initialize the GP with 25 random instances, and we demonstrate that even with random observations, AXE provides better tuning than any other recommended tuning method.

AXE Provides Practical Benefits for RocksDB. To analyze AXE's performance on a physical database, we deploy our tunings on a custom RocksDB instance where we implement the KLSM compaction policy. For every workload, we build a database with the respective tuning, load $5GB$ of data, and execute $100K$ operations following each workload distribution. Figure 10 shows the performance of each tuning method on a select number of workloads. Note that when tuning BO, each round executes a smaller subset of 1K operations that follows the distribution of the original workload; otherwise, the training cost would be orders of magnitude higher than the execution time. We observe that AXE recommends better-performing designs than all baselines in all workloads except

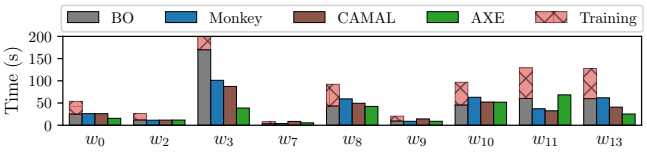


Figure 10: Comparison of execution time for tunings deployed on RocksDB for various baselines and AXE. Tunings recommended based on the cost model translate to the same performance on RocksDB.

w_{11}, which is dominated by read operations. CAMAL and Monkey provide better tunings as they target more bits per memory, while keeping the runs larger than those of AXE. However, both AXE and BO are navigating the larger design space of KSLM and, as such, are solving a harder problem. When we compare BO and AXE, we see that they are comparable in performance, differing by 10%, however, AXE does not need to pay the additional online training cost. Overall, AXE produces competitive tunings while requiring no online training time compared to BO, and generally outperforms the remaining baselines, particularly in terms of navigating a more complex design space.

Analyzing the Generalizability of AXE. In the following experiments, we analyze AXE's ability to generalize across design spaces and the associated cost. Here, we only compare AXE and BO, as they are not reliant on the cost model and can be extended to different design spaces by simply changing the training data. To analyze the cost of producing a tuning, we first define a measure that intuitively captures the cost of tuning. Because AXE is configured offline and can be queried to recommend tunings without additional overhead, no extra cost is incurred to produce a tuning at deployment time. Comparatively, BO is an iterative process that requires observations of the workload executions during each recommended tuning; hence, we measure BO's tuning efficiency as a function of the cost of deploying a tuning, captured by the cumulative regret R_ℓ over ℓ rounds. Hence, the cost of executing a workload using a tuning mainly affects the cumulative regret. We evaluate BO using cumulative regret R_ℓ^{BO} over ℓ rounds, defined as the difference between the cost of deploying an *unknown* optimal tuning and the cost of deploying the BO suggestion at every round i, or

$$R_\ell^{\text{BO}} = \sum_{i=1}^{\ell} \left(C_S(w, \varphi_i)\right) - C_S(w, \varphi^*) \tag{13}$$

Similarly, we measure the regret of AXE as the cost of deploying the tuning versus the optimal cost. Note that AXE is not an iterative method and therefore does not contain an online component. Once trained, the cost of procuring the best tuning from AXE is negligible. Therefore, the cumulative regret for AXE R_ℓ^{AXE} over ℓ rounds is

$$R_\ell^{\text{AXE}} = C_S(w, \varphi_{\text{AXE}}) - C_S(w, \varphi^*) \tag{14}$$

where φ_{AXE} is the tuning recommended by the learned tuner. We use both notions of regret to measure the relative cost of obtaining a tuning as the normalized expected cumulative delta regret

$$\mathbb{E}[\Delta R_\ell] = \mathbb{E}\left[\frac{R_\ell^{\text{BO}} - R_\ell^{\text{AXE}}}{C_S(w, \varphi_{\text{AXE}})}\right] \tag{15}$$

Table 4: On average, BO pays over 100× the overhead cost to produce marginally better tunings in the KLSM design space, and AXE performs better over the majority of rounds.

Type	$\mathbb{E}[\Delta R_\ell]$	$\Upsilon(w, \varphi_{\text{AXE}}, \varphi^*_{\text{BO}})$	$\Upsilon(w, \varphi_{\text{AXE}}, \varphi_{\text{BO}}) > 0$
Uniform	103	-0.1%	92%
Unimodal	106	-9%	53%
Bimodal	102	-5%	61%
Trimodal	102	-2%	74%

where we take the expectation over multiple runs to account for the stochastic nature of BO. Intuitively, a positive value implies that BO pays an upfront cost scaled to the cost of the tuning of AXE for that tuning performance at round ℓ.

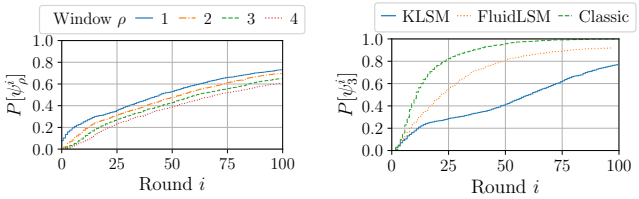
AXE Pays No Penalty for Flexibility. In Table 4 we show the expected normalized delta regret compared to the normalized performance difference for the best tuning produced by BO compared to AXE $\Upsilon(w, \varphi_{\text{AXE}}, \varphi^*_{\text{BO}})$ and the percentage of iterations where AXE outperforms BO for varying workload types. We observe that, on average, BO pays more than a 100× overhead cost to produce marginally better-performing tunings.

BO Tuning Performance Varies. We posit that BO outperforms AXE if it successively recommends tunings that outperform those recommended by AXE in a set number of consecutive rounds. Let ψ_ρ^i be the event that by round i, BO has recommended ρ successive tunings such that they outperform those recommended by AXE. Figure 11a shows the empirical cumulative distribution function (CDF) across multiple window sizes for the KLSM design space. We observe that the tunings from BO are less stable than those from AXE. As the window size increases, the probability that BO suggests consecutive tunings that perform better than AXE decreases. The online exploratory nature of GP continually tweaks suggested tunings in anticipation of further performance improvements, accumulating regret, thus requiring manual termination. In contrast, as AXE is trained offline, fine-tuning for performance improvements does not add additional regret.

AXE Scales With Tuning Knobs. Figure 11b shows the empirical CDF for a window size $\rho = 3$ for each design space – classic (Tiering vs Leveling), FluidLSM (last level, all levels above), and KLSM. BO quickly learns higher-performing tunings in the classic LSM design space, with 80% of runs recommending 3 better tunings by round 20. Similarly, BO effectively navigates the design space of FluidLSM; however, in the more complex space of KLSM, the probability BO outperforming AXE drops to 25% by round 20. This indicates that AXE scales to a larger number of tuning knobs and the complexity of the KLSM design space more effectively than BO.

6 DISCUSSION

Learning a Surrogate Cost Function is Easy. This work proposes a new learned tuning paradigm that is lightweight, trainable offline, and learns from historical data. We achieve this through a task decomposition approach [49] that divides the tuning problem into two subtasks: learning a surrogate cost model and navigating the cost surface. Training the models occurs offline, where any collected data points can be used, including analytical cost models



(a) Increasing window sizes **(b) Varying designs spaces**

Figure 11: The empirical CDF of ψ_ρ^i, defined as the round i where BO suggests ρ successive tunings that outperform AXE. BO is noisy; therefore, the tuning performance is unstable as the window size grows. AXE outperforms BO as the design space complexity (i.e., the number of tuning knobs) increases.

or historical workload executions. This enables the learned cost model to act as a surrogate for estimating execution cost.

Offline Task Decomposition Creates Better Tuners. Training our learned tuner occurs offline, where the learned cost model further augments its training dataset. The key benefit of training the tuner offline is that it removes the requirement for expensive experimentation at runtime (e.g., deploying and running a configuration); instead, we sample points from the learned cost model. This process requires significantly less domain knowledge than expertise-reliant techniques. Additionally, once trained, the cost for AXE to provide tuning suggestions is marginal compared to iterative techniques.

Weighing the Benefits of Retuning. For LSM tree-based storage engines, deploying a tuning can be a costly process, as different settings of high-impact knobs may fundamentally change the underlying data layout. For example, reducing the size ratio will require compactions throughout the entire tree, or changing the bits-per-element in the Bloom filters would require scanning the entire dataset and creating new SST files. As a result, applying a new tuning comes at a *retuning cost* which must be taken into account as part of the retuning decision, which is partially done by recently proposed online LSM tuning methods [54]. AXE is an offline solver that assumes a static LSM design, which cannot continually morph between tunings. However, changepoint detection [5] can be used to detect workload shifts and periodically query AXE to recommend new optimal tunings. Since this is a departure from our core objective of efficiently navigating the LSM cost surface and assumes a different design, we leave online LSM tuning as future work.

7 RELATED WORK

Auto-Tuning & Self-Designing Systems. The process of finding a system's optimal configuration has been extensively studied [11, 12], and traditionally focused on optimal index selection through an offline approach [10], where indexes are determined before deployment; an iterative online approach [48], with periodic index rebuilding, or an adaptive tuning approach [26], where indexes are incrementally rebuilt every query. Beyond the physical design of databases, auto-tuning has expanded to include optimal knob selection for both DBMS [27] and other data systems [50]. The tuning policies typically rely on user-defined cost models, rules-based methods [1, 10], or Bayesian Optimization techniques [2, 30, 33, 44, 57]. Zhang et al. highlight the advantages of Hyper Parameter Optimization methods like SMAC due to its ability to handle categorical parameters [56], while CDBTune and QTune use reinforcement learning for adaptive tuning based on live feedback [33, 55].

Self-designing systems further generalize auto-tuning by navigating a broader physical design space to build an optimal data system – rather than merely a database configuration – for specific applications [9, 25, 27, 28]. Similar to auto-tuning, self-designing systems typically define tuning policies via cost model analysis or machine learning methods While these prior works provide ways to refine or constrain the optimization process, AXE differs in that it is an offline process, effectively eliminating any online component and splitting the problem into separate learning tasks.

Learned Cost Models. Prior work on learned cost models focuses primarily on predicting the potential cost of queries to select the best plan to execute on the physical database [51–53, 58]. For database execution cost, Zero-Shot Cost Models are trained on various query executions and RDBMS generalize to unseen databases out-of-the-box [22]. AXE's task decomposition approach can be used to improve the estimation and navigation of query plans, and can integrate zero-shot models as its learned cost model, thus extending to other systems beyond LSM-based ones

Tuning LSM Trees. Prior work on LSM tuning with cost models primarily focuses on algorithms to optimize different filter structures for better read performance [14, 15], or tuning compaction policies in three spaces: Classic Tiering vs Leveling, FluidLSM [16], and KLSM [24, 35, 42]. RusKey [35] uses RL to tune LSM compaction based on workload shifts, while Smoose [42] applies dynamic programming to balance LSM trade-offs. Camal introduces an online active learning model to tune knobs at runtime [54]. ENDURE introduces a different paradigm by changing the LSM tuning problem to consider uncertainty during the tuning process [24]. WiscKey separates the physical storage layout of keys and values to prevent unnecessary data movement during compaction [38]. However, AXE remains the first learned tuning paradigm that tunes multiple knobs across various scenarios with minimal overhead.

8 CONCLUSION

In this work, we explore a task decomposition strategy for the LSM tuning problem and introduce AXE, an offline dual neural network approach to learning both subtasks of the problem. AXE first learns the cost of executing workloads on LSM configurations. This neural network is then used as the loss function of another model, termed *learned tuner*, that recommends high-performing tunings based on input workloads. Compared to a domain-expert-configured tuning pipeline, AXE requires significantly less prior expertise in modeling the system and in the optimizer's mechanics while recommending near-optimal tunings. Additionally, compared to online iterative strategies, we demonstrate that AXE is more efficient and incurs lower upfront costs while suggesting high-performing tunings. Overall, AXE is an easy-to-adopt and competitive strategy for recommending high-performance tunings at scale.

ACKNOWLEDGMENTS

This work is partially funded by the National Science Foundation under Grants No. IIS-2144547 and CCF-2403012, a Facebook Faculty Research Award, and a Meta Gift.

REFERENCES

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

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

[3] Sami Alabed and Eiko Yoneki. 2021. High-Dimensional Bayesian Optimization with Multi-Task Learning for RocksDB. In *Proceedings of the Workshop on Machine Learning and Systems (EuroMLsys)*. 111–119. https://doi.org/10.1145/3437984.3458841

[4] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1905–1916. http://www.vldb.org/pvldb/vol7/p1905-alsubaiee.pdf

[5] Matías Altamirano, François-Xavier Briol, and Jeremias Knoblauch. 2023. Robust and Scalable Bayesian Online Changepoint Detection. In *Proceedings of the International Conference on Machine Learning (ICML)*. 642–663. https://proceedings.mlr.press/v202/altamirano23a.html

[6] Apache. 2023. Cassandra. *http://cassandra.apache.org* (2023).

[7] Christopher M. Bishop and Hugh Bishop. 2024. *Deep Learning - Foundations and Concepts*. https://doi.org/10.1007/978-3-031-45468-4

[8] Stephen P Boyd and Lieven Vandenberghe. 2014. *Convex Optimization*. Cambridge University Press. https://web.stanford.edu/~boyd/cvxbook/

[9] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. 2021. Cosine: A Cloud-Cost Optimized Self-Designing Key-Value Storage Engine. *Proceedings of the VLDB Endowment* 15, 1 (2021), 112–126.

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

[11] Surajit Chaudhuri and Vivek R Narasayya. 2007. Self-Tuning Database Systems: A Decade of Progress. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 3–14. http://www.vldb.org/conf/2007/papers/special/p3-chaudhuri.pdf

[12] Surajit Chaudhuri and Gerhard Weikum. 2006. Foundations of Automated Database Tuning. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 104. http://vldbarc.org/archives/website/2006/slides/T1.pdf

[13] Carlo Curino, Neha Godwal, Brian Kroth, Sergiy Kuryata, Greg Lapinski, Siqi Liu, Slava Oks, Olga Poppe, Adam Smiechowski, Ed Thayer, Markus Weimer, and Yiwen Zhu. 2020. MLOS: An Infrastructure for Automated Software Performance Engineering. In *Proceedings of the Workshop on Data Management for End-To-End Machine Learning (DEEM@SIGMOD)*. 3:1–3:5. https://doi.org/10.1145/3399579.3399927

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

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

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

[17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 205–220. http://dl.acm.org/citation.cfm?id=1323293.1294281

[18] Facebook. 2024. RocksDB. *https://github.com/facebook/rocksdb* (2024).

[19] Johannes Freischuetz, Konstantinos Kanellis, Brian Kroth, and Shivaram Venkataraman. 2025. TUNA: Tuning Unstable and Noisy Cloud Applications. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*. 954–973. https://doi.org/10.1145/3689031.3717480

[20] Ian J Goodfellow, Yoshua Bengio, and Aaron C Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org/

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

[22] Benjamin Hilprecht and Carsten Binnig. 2022. One Model to Rule them All: Towards Zero-Shot Learning for Databases. https://arxiv.org/abs/2105.00642

[23] Andy Huynh, Harshal A. Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2022. Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1605–1618. https://doi.org/10.14778/3529337.3529345

[24] Andy Huynh, Harshal A. Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2024. Towards flexibility and robustness of LSM trees. *The VLDB Journal* (2024), 1–24. https://doi.org/10.1007/s00778-023-00826-9

[25] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, and Zichen Zhu. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. https://www.cidrdb.org/cidr2019/papers/p143-idreos-cidr19.pdf

[26] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. https://www.cidrdb.org/cidr2007/papers/cidr07p07.pdf

[27] Stratos Idreos and Tim Kraska. 2019. From Auto-tuning One Size Fits All to Self-designed and Learned Data-intensive Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. https://dl.acm.org/doi/abs/10.1145/3299869.3314034

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

[29] Eric Jang, Shixiang Gu, and Ben Poole. 2017. Categorical reparameterization with gumbel-softmax. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[30] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. 2022. LlamaTune: Sample-Efficient DBMS Configuration Tuning. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2953–2965. https://www.vldb.org/pvldb/vol15/p2953-kanellis.pdf

[31] Brian Kroth, Sergiy Matusevych, Rana Alotaibi, Yiwen Zhu, Anja Gruenheid, and Yuanyuan Tian. 2024. MLOS in Action: Bridging the Gap Between Experimentation and Auto-Tuning in the Cloud. In *Proceedings of the VLDB Endowment*. 4269–4272.

[32] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* (2015), 436.

[33] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2118–2130. http://www.vldb.org/pvldb/vol12/p2118-li.pdf

[34] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 2016. Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 149–166. https://www.usenix.org/conference/fast16/technical-sessions/presentation/lim

[35] Junfeng Liu, Fan Wang, Dingheng Mo, and Siqiang Luo. 2024. Structural Designs Meet Optimality: Exploring Optimized LSM-tree Structures in a Colossal Configuration Space. *Proceedings of the ACM on Management of Data (PACMMOD)* 2, 3 (2024), 1–26. https://doi.org/10.1145/3654978

[36] Liyuan Liu, Chengyu Dong, Xiaodong Liu, Bin Yu, and Jianfeng Gao. 2023. Bridging Discrete and Backpropagation: Straight-Through and Beyond. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*. http://papers.nips.cc/paper%5C_files/paper/2023/hash/28b5dfc51e5ae12d84fb7c6172a00df4-Abstract-Conference.html

[37] Ilya Loshchilov and Frank Hutter. 2017. SGDR: Stochastic gradient descent with warm restarts. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[38] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 133–148. https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu

[39] Chen Luo and Michael J. Carey. 2018. LSM-based Storage Techniques: A Survey. *CoRR* abs/1812.0 (2018). https://arxiv.org/abs/1812.07527

[40] Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. 2017. The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables. In *Proceedings of the International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=S1jE5L5gl

[41] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3217–3230. http://www.vldb.org/pvldb/vol13/p3217-matsunobu.pdf

[42] Dingheng Mo, Fanchao Chen, Siqiang Luo, and Caihua Shan. 2023. Learning to Optimize LSM-trees: Towards A Reinforcement Learning based Key-Value Store for Dynamic Workloads. *CoRR* abs/2308.0 (2023). https://doi.org/10.48550/arXiv.2308.07013

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

[44] David Buchaca Prats, Felipe Albuquerque Portella, Carlos Costa, and Josep Lluis Berral. 2020. You Only Run Once: Spark Auto-Tuning From a Single Run. *IEEE Transactions on Network and Service Management (TNSM)* 17, 4 (2020), 2039–2051. https://doi.org/10.1109/TNSM.2020.3034824

[45] RocksDB. 2021. RocksDB Tuning Guide. *https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide* (2021).

[46] Sebastian Sager, Hans Georg Bock, and Moritz Diehl. 2012. The integer approximation error in mixed-integer optimal control. *Math. Program.* 133, 1-2 (2012), 1–23. https://doi.org/10.1007/s10107-010-0405-3

[47] Subhadeep Sarkar, Kaijie Chen, Zichen Zhu, and Manos Athanassoulis. 2022. Compactionary: A Dictionary for LSM Compactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2429–2432.

[48] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. 2006. COLT: Continuous On-Line Database Tuning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 793–795. http://dl.acm.org/citation.cfm?id=1142473.1142592

[49] Alen D Shapiro. 1987. *Structured induction in expert systems*. Turing Institute Pr.

[50] Yu Shen, Xinyuyang Ren, Yupeng Lu, Huaijun Jiang, Huanyong Xu, Di Peng, Yang Li, Wentao Zhang, and Bin Cui. 2023. Rover: An Online Spark SQL Tuning Service via Generalized Transfer Learning. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 4800–4812. https://doi.org/10.1145/3580305.3599953

[51] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 99–113. https://doi.org/10.1145/3318464.3380584

[52] Ji Sun and Guoliang Li. 2019. An end-to-end learning-based cost estimator. *Proceedings of the VLDB Endowment* 13, 3 (2019), 307–319. https://doi.org/10.14778/3368289.3368296

[53] Jiani Yang, Sai Wu, Dongxiang Zhang, Jian Dai, Feifei Li, and Gang Chen. 2023. Rethinking Learned Cost Models: Why Start from Scratch? *Proceedings of the ACM on Management of Data (PACMMOD)* 1, 4 (2023), 1–27. https://doi.org/10.1145/3626769

[54] Weiping Yu, Siqiang Luo, Zihao Yu, and Gao Cong. 2024. CAMAL: Optimizing LSM-trees via Active Learning. *Proceedings of the ACM on Management of Data (PACMMOD)* 2, 4 (2024), 1–26. https://doi.org/10.1145/3677138

[55] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 415–432. https://doi.org/10.1145/3299869.3300085

[56] Xinyi Zhang, Zhuo Chang, Yang Li, Hong Wu, Jian Tan, Feifei Li, and Bin Cui. 2022. Facilitating Database Tuning with Hyper-Parameter Optimization: A Comprehensive Experimental Evaluation. *Proceedings of the VLDB Endowment* 15, 9 (2022), 1808–1821. https://www.vldb.org/pvldb/vol15/p1808-cui.pdf

[57] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2102–2114. https://doi.org/10.1145/3448016.3457291

[58] Johan Kok Zhi Kang, Gaurav Gaurav, Sien Yi Tan, Feng Cheng, Shixuan Sun, and Bingsheng He. 2021. Efficient Deep Learning Pipelines for Accurate Cost Estimations Over Large Scale Query Workload. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1014–1022. https://doi.org/10.1145/3448016.3457546

[59] Zichen Zhu, Ju Hyoung Mun, Aneesh Raman, and Manos Athanassoulis. 2021. Reducing Bloom Filter CPU Overhead in LSM-Trees on Modern Storage Devices. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*. 1:1–1:10.