
# MFTune: An Efficient Multi-fidelity Framework for Spark SQL Configuration Tuning

Beicheng Xu*†
Peking University
Beijing, China
beichengxu@stu.pku.edu.cn

Lingching Tung*†
Peking University
Beijing, China
lingchingtung@stu.pku.edu.cn

Yuchen Wang†
Peking University
Beijing, China
wychen@stu.pku.edu.cn

Yupeng Lu†
Peking University
Beijing, China
xinkelyp@pku.edu.cn

Bin Cui†
Peking University
Beijing, China
bin.cui@pku.edu.cn

## ABSTRACT

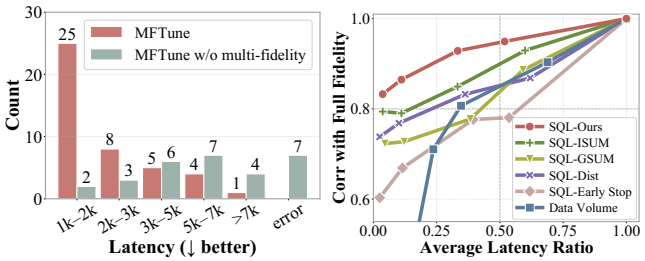
Apache Spark SQL is a cornerstone of modern big data analytics. However, optimizing Spark SQL performance is challenging due to its vast configuration space and the prohibitive cost of evaluating massive workloads. Existing tuning methods predominantly rely on full-fidelity evaluations, which are extremely time-consuming, often leading to suboptimal performance within practical budgets. While multi-fidelity optimization offers a potential solution, directly applying standard techniques—such as data volume reduction or early stopping—proves ineffective for Spark SQL as they fail to preserve performance correlations or represent true system bottlenecks. To address these challenges, we propose MFTune, an efficient multi-fidelity framework that introduces a query-based fidelity partitioning strategy, utilizing representative SQL subsets to provide accurate, low-cost proxies. To navigate the huge search space, MFTune incorporates a density-based optimization mechanism for automated knob and range compression, alongside an adapted transfer learning approach and a two-phase warm start to further accelerate the tuning process. Experimental results on TPC-H and TPC-DS benchmarks demonstrate that MFTune significantly outperforms five state-of-the-art tuning methods, identifying superior configurations within practical time constraints.

**PVLDB Reference Format:**
Beicheng Xu, Lingching Tung, Yuchen Wang, Yupeng Lu, and Bin Cui. MFTune: An Efficient Multi-fidelity Framework for Spark SQL Configuration Tuning. PVLDB, 19(11): 3649 - 3662, 2026.
doi:10.14778/3836663.3836715

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/PKU-DAIR/MFTune.

*Both authors contributed equally to this research.
†All authors are with the School of Computer Science & Key Lab of High Confidence Software Technologies (MOE) & Beijing Key Laboratory of Software and Hardware Cooperative Artificial Intelligence Systems, Peking University, Beijing, China.


Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836715




(a) Performance distribution. (b) Fidelity correlation.

**Figure 1: Analysis of multi-fidelity mechanism on TPC-DS (600GB). (a) Latency distribution of configurations evaluated within 48 hours. (b) Fidelity correlation. We evaluate six proxies: 1) SQL-Ours, selecting SQL subsets from $\frac{1}{27}$ to 1 fidelity based on Section 7.1; 2) SQL-ISUM, 3) SQL-GSUM, and 4) SQL-Dist, adapted from ISUM [50], GSUM [8], and the distance-based method [6] to our setting, respectively, detailed in Section 8.4.1; 5) SQL-Early Stop, executing prefixes from $\frac{1}{27}$ to 1 of the total SQLs; and 6) Data Volume, scaling the dataset across {30,100,200,400,600} GB. Each point reports the Kendall correlation between proxy and full-fidelity latency versus the average latency ratio, computed over 50 configurations.**

## 1 INTRODUCTION

The recent explosion of data has catalyzed the widespread deployment of large-scale query systems, such as Hive [54], Presto [47], and Spark SQL [2]. Among these, Spark SQL has emerged as a premier choice for industry-scale data processing due to its robust integration with the Spark ecosystem [63] and its superior performance in handling complex analytical workloads [21]. In real-world scenarios, Spark SQL workloads typically have two characteristics. First, they often include multiple queries, as production data processing tasks are commonly structured as multi-stage workflows, such as Extract, Load, and Transform pipelines in data warehousing [24] and feature generation for machine learning [17]. Second, they typically process large volumes of data, spanning hundreds of gigabytes to several terabytes or more [59]. The combination of multiple queries and massive datasets makes Spark SQL workloads

highly resource-intensive, often requiring hours or even days to complete. Meanwhile, the efficiency of Spark SQL is heavily dependent on more than 200 configuration knobs that govern system behaviors like memory management and shuffle partitioning [52]. Because default settings are often suboptimal, each evaluation is costly, and the configuration space is prohibitively large, manual tuning is both time-consuming and far from optimal.

Therefore, various configuration tuning methods have been proposed to find optimal configurations for Spark SQL workloads automatically [7, 11, 33, 36, 48, 62]. These methods typically adopt the Bayesian Optimization (BO) framework [12, 16, 51] to sample promising configurations, iteratively replay the workload, and model the relationship between system performance and parameters. However, these methods suffer from several fundamental challenges when applied to Spark SQL, as follows:

**C1. Prohibitive evaluation costs.** Given the long-running nature of Spark SQL workloads, evaluating configurations is highly expensive. However, existing tuning methods often rely on full-fidelity evaluations (executing the entire original workload) [11, 33, 36], which limits the number of configurations that can be explored within a finite tuning budget. A potential solution to mitigate this bottleneck is to adopt Multi-Fidelity Optimization (MFO) [10, 34, 39]. MFO leverages low-fidelity evaluations, which are faster but approximate the full-fidelity results, to efficiently filter out poor configurations and guide the search toward optimal regions. Our preliminary analysis in Figure 1a demonstrates the impact of this mechanism: within a 48-hour budget, MFTune successfully evaluated 43 configurations with MFO. In contrast, a full-fidelity variant (MFTune w/o multi-fidelity) evaluated only 29 configurations in the same time frame, with the majority falling into higher latency buckets or even resulting in execution errors. This comparison illustrates that relying solely on full-fidelity evaluations makes it infeasible to explore the vast search space effectively within practical budgets.

**C2. Inapplicability of directly applying existing MFO methods.** The core challenge of MFO lies in ensuring that the low-fidelity proxies preserve the relative ranking of configurations observed under full fidelity. However, existing standard fidelity-partitioning techniques used in Automated Machine Learning (AutoML), such as training on a subset of data [25, 38, 39] or for fewer epochs [10, 20, 38], do not transfer effectively to Spark SQL. As illustrated in Figure 1b, generic low-fidelity proxies in AutoML fail to maintain a reliable correlation with full-fidelity performance in Spark SQL. Reducing data volume leads to a sharp decline in correlation because smaller datasets fail to trigger the true bottlenecks at full scale. Beyond generic proxies, we also adapt DBMS workload-compression methods [6, 8, 50]. However, the same figure shows that they still provide weak correlation with full-fidelity as they are designed for different downstream objectives like index tuning rather than preserving configuration rankings for Spark SQL MFO. In contrast, our proposed SQL Selection maintains a high Kendall-tau correlation (>0.8) even at extremely low fidelity levels. These findings demonstrate that customizing fidelity partitioning for the Spark SQL domain is essential, as neither generic AutoML proxies nor DBMS workload-compression methods can provide the reliable lightweight feedback required for effective tuning.

**C3. Huge search space and ineffective compression.** The search space of Spark SQL is prohibitively large for exhaustive exploration within a limited budget. Most existing works focus solely on reducing the number of knobs via sensitivity analysis or projection [11, 36, 60, 62]. Only a few methods [53, 68] have attempted to compress the value range. However, their bucket- or boundary-based strategies are sensitive to outliers and may miss promising but underexplored regions. Consequently, robust and fine-grained search-space compression remains an open challenge.

To address these challenges, we propose MFTune, an efficient multi-fidelity framework tailored for Spark SQL configuration tuning. Our key contributions are summarized as follows:

- To the best of our knowledge, we are the first to introduce multi-fidelity optimization (MFO) into Spark SQL tuning. We further tailor transfer learning and warm-start mechanisms to accommodate varying fidelity levels.
- We propose a query-based multi-fidelity partitioning algorithm to select representative SQL query subsets that maintain a high performance correlation with full-fidelity workloads.
- We introduce a unified framework for knob selection and range compression based on weighted density modeling, which captures high-potential regions accurately.
- Extensive evaluations across various benchmarks, hardwares, and data scales demonstrate the superiority of MFTune over state-of-the-art baselines, yielding 25.9%–43.1% and 37.8%–69.3% latency reduction on TPC-H and TPC-DS, respectively.

## 2 RELATED WORK

### 2.1 Automatic System Configuration Tuning

**DBMS tuning**. Various automatic configuration tuning methods have been proposed for Database Management Systems (DBMS) [15, 31, 69]. Among them, Bayesian Optimization (BO)-based methods and Reinforcement Learning (RL)-based methods have been the most widely used. BO-based methods (e.g., OtterTune [57], ResTune [66], LlamaTune [23], OpAdvisor [68], TopTune [60]) iteratively sample configurations and replay the workload to model the relation between configuration and performance. RL-based methods (e.g., CDBTune [64], QTune [30], SHA [32], UDO [58]) use RL to predict the reward of a configuration, and explore new configurations to maximize the reward, which requires more resources to converge. Lero [70] uses a pairwise ranker to select execution plans for query optimization. Recent studies have also leveraged LLMs for DBMS tuning. DB-BERT [56] extracts tuning hints from manuals and documents, while GPTuner [28], Rabbit [53], and AgentTune [37] use LLMs or LLM agents to exploit textual knowledge for knob selection, range pruning, and configuration optimization. Another line of studies focuses on the online setting and addresses the workload shift during the tuning process [49, 67]. In the noisy cloud environment, TUNA [13] improves robustness by evaluating configurations across machines and adjusting noisy measurements.

**Spark SQL Tuning**. While DBMS tuning methods can in principle be adapted to Spark SQL, the significantly longer query execution times in Spark SQL make methods that require hundreds of iterations [4, 66, 68] extremely time-consuming. Therefore, recent studies have developed Spark SQL-specific auto-tuning methods, most of which are based on BO and strive to enhance tuning efficiency to mitigate the evaluation overhead. For example, TIE [7]

accelerates the sampling phase of BO through an early termination mechanism, thereby reducing the tuning time. LOCAT [62] compresses the target workload after obtaining sufficient evaluation observations of the original workload, and performs BO in a reduced search space. To avoid tuning each workload from scratch, some methods leverage knowledge from similar historical tuning tasks [11, 33, 36, 48]. For example, Tuneful [11] employs a Multi-Task Gaussian Process to train and predict similar historical task data alongside current task observations. Online-Tune [36] and LOFTune [33] obtain optimal configurations from similar tasks for a warm start. Rover [48] accelerates tuning by weighting the BO acquisition function with historical tasks. Rockhopper [71] studies online Spark configuration tuning in production environments, focusing on robust adaptation to noisy and dynamic workloads. However, all the aforementioned methods share a common limitation: they rely on complete evaluations of the workload. Within a limited budget, it's challenging to evaluate enough configurations on complex, large-scale Spark SQL workloads, leading to insufficient search space exploration and reduced optimization efficiency. LOCAT [62] is the only approach that considers workload compression, but it requires extensive random search on the original workload to collect compression observations, which remains time-consuming. Overall, efficient Spark SQL configuration tuning remains a significant challenge.

## 2.2 Multi-Fidelity Optimization

Although system parameter tuning currently relies on complete evaluations, AutoML has been exploring multi-fidelity optimization to tackle this challenge [14, 20, 25, 34, 46]. Typical MFO methods use cheap low-fidelity evaluations to guide search: Hyperband [34] (HB) dynamically allocates resources to a set of random configurations, and uses the successive halving algorithm [18] to stop poor-performing configurations in advance. BOHB [10] and MFES-HB [39] improve HB by replacing random sampling with BO. ASHA [35], A-BOHB [27], and Hyper-Tune [38] extend HB to the asynchronous case when using multiple workers. An orthogonal line of work models the learning curves of machine learning algorithms directly [41]. These studies often use neural networks to rank learning curves across different configurations [26, 61]. MFIX [5] further adapts MFO to database index advising. It uses a what-if optimizer interface as the low-fidelity proxy, which is not available in Spark SQL knob tuning.

MFO provides an opportunity to reduce reliance on complete evaluations through cheaper proxies. An effective low-fidelity proxy should preserve full-fidelity configuration rankings, since MFO uses it to prune poor configurations early. However, existing MFO methods define low-fidelity proxies for their own domains and do not transfer well to Spark SQL tuning. As shown in Figure 1b, generic proxies such as data volume reduction or early stop have low correlation with the full-fidelity performance, making them unreliable for Spark SQL configuration tuning.

## 2.3 Workload Compression

The database community has long studied workload compression, which replaces a large SQL workload with a subset for downstream applications. Chaudhuri et al. [6] define application-specific query distances to bound the loss of replacing one query with another for index selection and approximate query processing. GSUM [8] constructs representative query subsets by matching feature distributions and improving feature coverage. ISUM [50] compresses workloads for scalable index tuning by selecting queries with high utility-influence benefit, where utility estimates index-induced cost reduction and influence is derived from indexable-column features.

These studies show the potential to reduce large workloads for downstream DBMS tasks. However, directly using them as Spark SQL fidelity proxies faces two mismatches: First, several required signals [6, 50], such as hypothetical-index benefits and AQP approximation errors, are unavailable in the Spark SQL scenario. Second, they select queries by optimizing feature coverage [8], or query-to-query influence [6, 50], rather than directly optimizing the configuration-ranking consistency between the subset and the full workload. This ranking consistency is essential in Spark SQL multi-fidelity tuning, where low-fidelity evaluations decide which configurations proceed to full-workload evaluation. Accordingly, Figure 1b shows that even adapted workload-compression methods still provide weak correlation with full-fidelity performance. Together with the limitations of generic MFO proxies, this underscores the need for a Spark SQL-specific fidelity partitioning strategy.

# 3 PRELIMINARIES

## 3.1 Problem Definition

Given a target workload $W$, which consists of multiple SQL queries over a database, the goal of Spark SQL configuration tuning is to find the optimal Spark configuration $\boldsymbol{x}^*$ that minimizes a metric:

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}\in\mathcal{X}} f(\boldsymbol{x}, W), \tag{1}$$

where $\mathcal{X}$ is the Spark configuration search space and $f(\boldsymbol{x}, W)$ is the performance (in our case, latency) of $W$ under configuration $\boldsymbol{x}$.

## 3.2 Bayesian Optimization Framework

Bayesian optimization (BO) [3, 16, 51] is a popular algorithm framework that solves black-box problems with a costly objective function. It loops over the following steps: 1) it fits a *surrogate model* $M$ based on observervations $D = \{(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_{n-1}, y_{n-1})\}$, where $\boldsymbol{x}_i$ is the $i^{th}$ configuration and $y_i$ is the corresponding observed performance; 2) with the surrogate $M$, BO chooses the next configuration by maximizing the *acquisition function* $\boldsymbol{x}_n = \arg\max_{\boldsymbol{x}\in\mathcal{X}} \alpha(\boldsymbol{x}; M)$; 3) It evaluates the configuration $\boldsymbol{x}_n$ to obtain its performance $y_n$ and augments the observations by $D = D \cup \{(\boldsymbol{x}_n, y_n)\}$.

**Surrogate**. We apply the Probabilistic Random Forest [16] as the surrogate. Compared to alternatives, it has lower time complexity and shows better empirical results in real-world scenarios [65].

**Acquisition function**. We apply Expected Improvement (EI) [22] to select candidate configurations. Given the surrogate predictive distribution $p_M(y|\boldsymbol{x})$, EI measures the expected gain over the best observed performance $y^*$: i.e., $EI(\boldsymbol{x}) = \int_{-\infty}^{\infty} \max(y - y^*, 0) p_M(y|\boldsymbol{x}) dy$.

## 3.3 Hyperband Framework

Hyperband (HB) [34] is a classic MFO framework that halts poorly performing configurations early. It consists of three key components: a fidelity partitioning strategy, and nested outer and inner loops responsible for scheduling multi-fidelity evaluations.

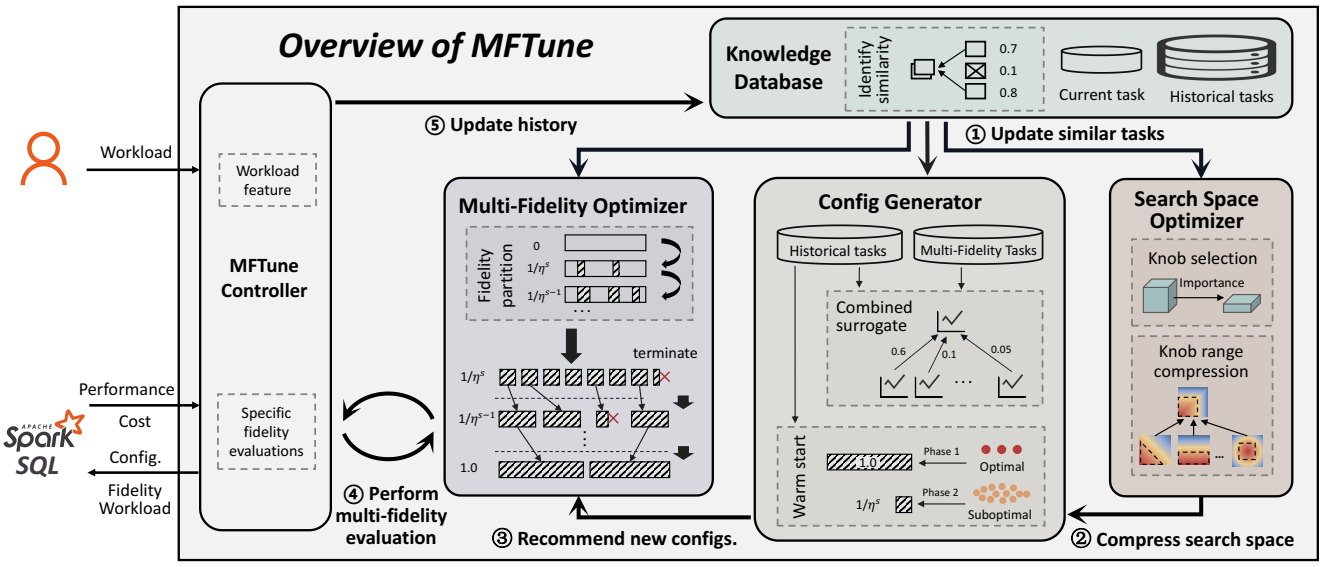



**Figure 2: Overview: architecture and workflow of MFTune.**

**Fidelity partition** determines how to allocate $r (r \leq R)$ units of resources to evaluate a configuration when the maximum resource is $R$. As existing methods are particularly applied in AutoML, the typical approaches involve either training for fewer epochs [10, 20, 38] or using a subset of the dataset [25, 38, 39].

**Inner loop: successive halving (SH).** Given a type of resource partition (e.g., number of epochs or dataset size), HB initially evaluates $n_1$ configurations with each using $r_1$ units of resources, ranking them by performance. HB then continues with only the top $\eta^{-1}$ configurations using $\eta$ times more resources (typically $\eta = 3$), i.e., $n_2 = n_1 \times \eta^{-1}$ and $r_2 = r_1 \times \eta$. This process repeats until the maximum training resource $R$ is reached, i.e., $r_i = R$.

**Outer loop: grid search of $n_1$ and $r_1$.** HB performs a grid search over $(n_1, r_1)$ pairs to cover different SH brackets. Specifically, HB enumerates a bracket index $s \in \{s_{\max}, s_{\max} - 1, \ldots, 0\}$, where $s_{\max} = \lfloor \log_\eta R \rfloor$. Each $s$ defines one feasible pair:

$$n_1 = \left\lceil \frac{\mathcal{B}}{R} \frac{\eta^s}{s+1} \right\rceil, \qquad r_1 = R\eta^{-s},$$

where $\mathcal{B} = (s_{\max} + 1)R$. Thus, a larger $s$ corresponds to more configurations with fewer initial resources, and vice versa.

## 4 SYSTEM OVERVIEW

Figure 2 provides an overview of MFTune, which consists of five components: 1) The controller interacts with users and the Spark SQL server, and controls the tuning process. 2) The knowledge database stores both observations and meta-features from the current task as well as historical tasks. 3) The search space optimizer identifies a promising subspace from the original search space. 4) The configuration generator suggests a set of promising configurations. 5) The multi-fidelity optimizer constructs fidelity levels and schedules evaluations across them.

**Workflow**. To initiate a tuning task, the user first specifies the tuning objective, the tuning budget, and uploads the target workload to the controller. An iterative tuning workflow is then activated: ① The **knowledge database** first estimates source-target task similarity using meta-features and historical observations, and provides similar source tasks to other components. ② The **search space optimizer** then compresses the original search space using observations from similar tasks, including both knob selection and range narrowing. ③ After that, the **configuration generator** proposes a batch of promising configurations from the compressed subspace. Among these, a subset is drawn from top-performing configurations of similar tasks for warm-start, while the rest is generated by Bayesian optimization using weighted surrogates from source tasks and low-fidelity observations. ④ Upon receiving the candidate configurations, the **multi-fidelity optimizer** iteratively evaluates and prunes them. If no fidelity partition has been built, it first derives one from similar tasks; this step is performed only once. It then starts from the lowest fidelity, progressively prunes poor configurations, and increases fidelity until reaching the full workload. ⑤ Finally, the **MFTune controller** collects the iteration results and stores them in the knowledge database. These five steps repeat until the user-specified budget, such as time or iteration limits, is exhausted. After that, MFTune selects the best configuration observed in the current task and returns it to the user.

The following sections follow this workflow: Section 5 describes step ① for identifying transferable source tasks; Section 6 describes step ② for configuration-space compression; and Section 7 describes steps ③–④, including candidate generation, query-based fidelity partitioning, and multi-fidelity optimization.

## 5 SIMILARITY IDENTIFICATION

This section describes how MFTune identifies historical tasks that are useful for the current tuning task.

A continuously deployed Spark SQL tuning system can accumulate historical tasks over time as it performs tuning tasks on different workloads. These previous tasks can offer valuable insights for the

current tuning. For example, workloads from different hardwares or data scales can expose critical queries for selecting representative SQL subsets. Even different-query workloads can provide useful signals for identifying promising search-space regions [48, 68]. The intuition is that different Spark SQL workloads can share system-level bottlenecks, such as shuffle parallelism, memory pressure, spill behavior, executor resource allocation, and CPU/I/O contention. When two workloads expose similar bottlenecks, configurations that affect these bottlenecks tend to preserve similar performance rankings, even if absolute latencies differ. However, as prior studies show, transferring knowledge from dissimilar tasks can lead to negative impacts [49]. Hence, for a given target task, MFTune first identifies similar source tasks for safe transfer.

Task similarity is commonly measured by whether two tasks induce similar configuration-performance rankings [53, 66]. Given the observations $\mathcal{D}_T$ of the target task, a source task is considered similar if its surrogate can correctly predict the relative performance of $\mathcal{D}_T$, indicating similar system bottlenecks between the two tasks. Therefore, we define the similarity $S(i, T)$ between the $i$-th source task and the target task using the Kendall-tau coefficient:

$$S(i, T) = \text{Kendall}^{\mathcal{D}_T}(M_i, Y) \tag{2}$$

where $M_i$ is the BO surrogate model trained on source observations $\mathcal{D}_i$, and $\text{Kendall}^{\mathcal{D}_T}(M_i, Y)$ is the Kendall-tau coefficient between the predictive values and the observed performance on $\mathcal{D}_T$.

**Warm-starting through prediction**. However, existing works often overlook a limitation of this similarity metric: when $\mathcal{D}_T$ is small, the Kendall-tau estimate in Equation 2 becomes unreliable, causing inaccurate and unstable similarity estimates in early tuning iterations. To mitigate this issue, we extend Rover [48] to warm-start similarity estimation with a similarity predictor. We first extract query-level features from SparkEventLog under the default configuration and average them as the task meta-feature. A LightGBM regressor is trained to predict pairwise task similarity from two task meta-features. It is trained on tuning histories in the knowledge database. For tasks $i$ and $j$, the label is $\text{Kendall}^{\mathcal{D}_{rand}}(M_i, M_j)$, the Kendall-tau coefficient between their surrogate predictions on random configurations. After that, for a new tuning task, the trained predictor provides warm-start similarities to all source tasks.

**Transition mechanism**. As for the transition from predicted similarity to Kendall-tau similarity computed from target observations, the key insight is that directly computing Equation 2 from only a few target observations can be unstable and may assign misleading weights to source tasks. Therefore, MFTune keeps using meta-feature-based predicted similarity in the early stage. At each iteration, MFTune also attempts to compute the Kendall-tau similarity in Equation 2 from the accumulated target observations, but switches to it only when the majority of source tasks have Kendall-tau $p$-values below 0.05. Satisfying this condition indicates that the accumulated target observations are sufficient for reliable similarity computation, making it less sensitive to small-sample fluctuations and closer to the true source-target relationship.

**Weighting based on similarity**. After obtaining the similarities of all source tasks, we first filter out source tasks with negative similarity, as they provide less utility for the target task than a random guess. Subsequently, the similarities are normalized as transfer weights, where the weight of source task $i$ is $w_i = S(i, T)/\sum_j S(j, T)$. For generality, MFTune also assigns a weight to the target tuning task. To achieve that, we measure its generalization ability on unseen configurations using the out-of-sample Kendall-tau coefficient. Specifically, we perform cross-validation on $\mathcal{D}_T$, training the target surrogate on the remaining folds to predict each held-out fold. After the transition mechanism enables Equation 2, the target task also receives a weight based on it. Overall, this weighting mechanism supports all transfer-related components in MFTune: It aggregates promising regions for search-space compression (Section 6), guides SQL-subset selection for fidelity partitioning (Section 7.1), and combines surrogates for candidate generation (Section 7.3)

## 6 CONFIGURATION SPACE COMPRESSION

To handle the huge Spark configuration space, it is necessary to compress the original space into a promising subspace to improve exploration efficiency [23, 60, 68]. Search-space compression includes knob selection and value-range narrowing. For knob selection, prior methods use sensitivity analysis, such as Lasso [55] and SHAP [40], or low-dimensional projections, such as HeSBO [29] and ALEBO [42] [11, 23, 36, 60, 62]. For value-range compression, a common strategy is to transfer promising regions from similar tasks [53, 68]. However, they often define ranges only by the boundaries of discrete historical configurations, which is sensitive to outliers and likely to miss promising but underexplored values. This section explains how MFTune compresses the search space by first extracting promising regions from each source task and then aggregating them into selected knobs and value ranges.

### 6.1 Promising Regions Extraction

MFTune first extracts the promising discretized region for each knob from similar source tasks. Specifically, for source task $i$ with observations $\mathcal{D}_i$, we calculate the median performance of all recorded configurations, denoted as $f^i_{\text{median}}$. The promising set $G_i$ contains configurations better than this median: i.e., $G_i = \{\boldsymbol{x} \in \mathcal{D}_i \mid f^i(\boldsymbol{x}) < f^i_{\text{median}}\}$, where $f^i(\boldsymbol{x})$ is the performance of $\boldsymbol{x}$ in the source task.

Although all configurations in $G_i$ are promising, not every knob value in these configurations necessarily contributes positively to the performance. Therefore, we employ SHAP to analyze the importance of each knob. We choose SHAP because it provides local, value-level, and sign-sensitive attribution, which allows MFTune to identify knob values that reduce latency. In contrast, Lasso, Gini importance, and fANOVA mainly provide global importance or scores, rather than signed contributions of specific knob values. Specifically, SHAP estimates the contribution of each knob value in every configuration of $G_i$. A positive SHAP value means the knob value increases latency, while a negative value means it reduces latency. We then define the promising knob values as those with negative SHAP values, assuming latency minimization. For knob $k_j$, its promising value set $P^i_j$ in source task $i$ is:

$$\begin{aligned} P^i_j &= \Big\{ \big(\boldsymbol{x}[k_j],\ v(\boldsymbol{x})\big) \mid \boldsymbol{x} \in G_i \text{ and } \text{SHAP}(\boldsymbol{x}[k_j]) < 0 \Big\}, \\ v(\boldsymbol{x}) &= w_i \cdot \frac{f^i_{\text{median}} - f^i(\boldsymbol{x})}{f^i_{\text{median}}}, \end{aligned} \tag{3}$$

where $\boldsymbol{x}[k_j]$ denotes the value of $k_j$ in configuration $\boldsymbol{x}$ and $v(\boldsymbol{x})$ is the weight of this value. We define $v(\boldsymbol{x})$ as the product of the

source task's weight $w_i$ and the relative performance improvement of configuration $\boldsymbol{x}$ over $f^i_{\text{median}}$, so values from better configurations in more similar tasks receive larger weights.

## 6.2 Density Modeling and Target Search Space

After extracting promising discrete regions from each source task, MFTune constructs the target search space. For each knob $k_j$, if the weighted majority of tasks provide no promising values, i.e., $\sum_{i=1}^{K} w_i \cdot \mathbb{I}(P^i_j = \emptyset) > 0.5$, MFTune removes this generally unpromising knob from the search space. Otherwise, MFTune constructs the final promising value set $P_j$ for $k_j$ by merging the promising values, i.e., $P_j = \bigcup_{i=1}^{K} P^i_j$. Instead of directly using the boundaries of the discrete historical values, MFTune estimates their empirical density to find regions where promising values concentrate. This smooths the promising-value evidence transferred from similar tasks.

For continuous knobs, we use **Kernel Density Estimation (KDE)** [44] to estimate the empirical density of their promising values. Specifically, for each continuous knob, we collect the values corresponding to promising configurations and use these samples, along with their associated weights, to construct a weighted density curve. The weighted KDE function is defined as:

$$\hat{g}(x) = \frac{1}{h \cdot \sum_{(\theta,v)\in P_j} v} \sum_{(\theta,v)\in P_j} v \cdot K\left(\frac{x-\theta}{h}\right), \tag{4}$$

where each $(\theta, v)$ pair represents a promising value and its weight, $h$ is the bandwidth determined by Silverman's rule of thumb, and $K(\cdot)$ is the kernel function, typically Gaussian. We use Silverman's rule as a lightweight default for setting the bandwidth; the resulting weighted sum of local kernels can reflect multiple concentrated regions in the empirical promising-value distribution. The weights make values from more promising configurations contribute more to the density curve, providing a smoother and more data-driven representation than the discrete boundaries used in [53, 68].

To compress the range of a continuous knob, MFTune selects the densest region from the estimated density $\hat{g}(x)$. We define a cumulative density threshold $\alpha \in [0, 1]$ as the minimum proportion of total probability mass to retain. This threshold controls the trade-off between compression strength and conservativeness: a smaller $\alpha$ retains keeps only the densest core regions, while a larger $\alpha$ preserves more candidate regions to reduce the risk of over-pruning. Assume the original range of $k_j$ is $R_j \subset \mathbb{R}$, the goal is to identify the smallest sub-region $R^{\text{opt}}_j$ that covers at least $\alpha$ probability mass:

$$\min_{R^{\text{opt}}_j \subset R_j} \quad |R^{\text{opt}}_j| \qquad \text{s.t.} \int_{R^{\text{opt}}_j} \hat{g}(x)\, dx \geq \alpha, \tag{5}$$

In practice, MFTune sorts density values in descending order and accumulates them until the retained probability mass reaches at least the minimum proportion $\alpha$. The corresponding values define the compressed range $R^{\text{opt}}_j$.

To process categorical knobs, we redefine Equation 4 in a discrete form to build the density function with the indicator function $\mathbb{I}(\cdot)$:

$$\hat{g}(x) = \frac{1}{\sum_{(\theta,v)\in P_j} v} \sum_{(\theta,v)\in P_j} v \cdot \mathbb{I}(x = \theta). \tag{6}$$

The remaining steps follow the continuous-knob case, yielding a unified framework for categorical and continuous knobs.

Compared to existing methods, our approach offers three advantages: 1) It does not require manually setting the number of retained knobs. Through aggregating promising regions from similar tasks, it performs knob selection and range compression jointly. 2) Density modeling captures concentrated promising regions, making compression less sensitive to outliers and able to retain nearby underexplored values. 3) As the similarity estimate of source tasks become more accurate during tuning, the compressed search space can be updated dynamically.

# 7 MULTI-FIDELITY OPTIMIZATION

Following similarity identification and space compression, MFTune MFTune applies MFO within the subspace. Specifically, this section introduces how MFTune 1) partitions fidelity levels, 2) generates candidate configurations, and 3) executes the MFO process.

## 7.1 Fidelity Partition

As discussed in Section 3.3, fidelity partitioning is central to MFO. We treat evaluation on the original workload as full fidelity, e.g., running 99 queries on 1TB of data. A $\delta$-fidelity evaluation ($0 < \delta \leq 1$) uses a $\delta$ fraction of the full-fidelity resource cost, such as execution time or memory consumption. For effective pruning in MFO, low-fidelity evaluations should preserve the relative ranking of configurations, so configurations that perform well at full fidelity are not mistakenly discarded. Since existing low-fidelity choices, such as smaller datasets, early stopping, and database what-if interfaces, are unreliable or unavailable for Spark SQL tuning, MFTune constructs Spark SQL-specific fidelities by selecting query subsets.

Assume that the original workload consists of a set of queries $Q = \{q_1, q_2, \ldots, q_m\}$. Fidelity partitioning aims to select a representative subset $Q_\delta \subseteq Q$ that preserves the full-workload evaluation ranking while reducing evaluation cost. Specifically, for $\delta$-fidelity evaluation, the selected subset $Q_\delta$ should satisfy a cost constraint $\text{Cost}(Q_\delta) \leq \delta \cdot \text{Cost}(Q)$. The problem is formulated as:

$$\max_{Q_\delta \subseteq Q} \ \tau(Q_\delta, Q) \quad \text{s.t.} \quad \text{Cost}(Q_\delta) \leq \delta \cdot \text{Cost}(Q), \tag{7}$$

where $\tau(Q_\delta, Q)$ measures the correlation between the subset $Q_\delta$ and $Q$. Given per-query performance across configurations, we compute $\tau$ using rank-based metrics such as Kendall-tau.

However, directly estimating $\tau$ on the target task requires sufficient full-fidelity evaluations, which is prohibitively expensive. To address this, MFTune transfers evidence from source tasks with the same query workload but different execution conditions. Such same-query source tasks naturally arise in practice, including: 1) *data-scale evolution*, where the same query set is retuned as data grows or is refreshed; 2) *hardware/resource migration*, where the workload moves across clusters, instance types, or CPU/memory settings; and 3) *system performance testing*, where teams repeatedly run the same analytical workload for regression testing, benchmarking, or capacity planning. In these scenarios, the query set remains stable while data scale, hardware, Spark/runtime version, or resource environment changes. Therefore, these source tasks can provide evidence for selecting representative SQL subsets without extensive new full-fidelity evaluations on the target task. Moreover, transferring critical-query evidence from similar source tasks is also

**Algorithm 1** Greedy Solution for Query Subset Selection

**Input:** Full query set $Q$, fidelity $\delta$, source observations $\{\mathcal{D}_i\}_{i=1}^{K'}$
**Output:** Optimal subset $Q_\delta$

```
Initialize Q_δ ← ∅ and cost ratio r ← 0 {Start with empty}
Compute the weighted average cost fraction of each query:
  c(q) = Σ_i w_i · (Σ_{x∈C_i} c_{q,x}) / (Σ_{q_j∈Q} Σ_{x∈C_i} c_{q_j,x}).
while True do
  q* ← arg max_{q∈Q\Q_δ, r+c(q)≤δ} τ(Q_δ ∪ {q}, Q)
  if q* is null then break
  Q_δ ← Q_δ ∪ {q*}, r ← r + c(q*)
end while
return Q_δ
```

reasonable. Task similarity implies that configurations have consistent relative performance across tasks, indicating shared system bottlenecks. This consistency suggests that queries exposing key bottlenecks in one task are likely important in another. Therefore, we first select source tasks with the same query set as the target task. For the $i$-th source task, let $\mathcal{D}_i = \{(\boldsymbol{x}, \mathbf{p}_{\boldsymbol{x}}, \mathbf{c}_{\boldsymbol{x}})\}_{\boldsymbol{x}\in C_i}$ denote its observations, where $C_i$ is the evaluated configuration set. Here, $\mathbf{p}_{\boldsymbol{x}} = [p_{q_1,\boldsymbol{x}}, p_{q_2,\boldsymbol{x}}, \ldots, p_{q_m,\boldsymbol{x}}]$ records the per-query performance of configuration $\boldsymbol{x}$, and $\mathbf{c}_{\boldsymbol{x}} = [c_{q_1,\boldsymbol{x}}, c_{q_2,\boldsymbol{x}}, \ldots, c_{q_m,\boldsymbol{x}}]$ records the corresponding per-query costs. Commonly, an aggregation function $\mathrm{Agg}(\cdot)$ combines per-query performance into workload-level performance for a query set. For example, the latency metric can be aggregated by summation, i.e., $\mathrm{Agg}(Q_{\mathrm{var}}, \mathbf{p}_{\boldsymbol{x}}) = \sum_{q\in Q_{\mathrm{var}}} p_{q,\boldsymbol{x}}$, where $Q_{\mathrm{var}}$ is a query set. The correlation score of $Q_\delta$ in source task $i$ is:

$$\begin{aligned} \tau_i(Q_\delta, Q) &= \mathrm{KendallTau}(A_i^{Q_\delta},\ A_i^{Q}), \\ A_i^{Q_{\mathrm{var}}} &= \left\{\mathrm{Agg}(Q_{\mathrm{var}}, \mathbf{p}_{\boldsymbol{x}}) \mid \boldsymbol{x} \in C_i\right\}. \end{aligned} \tag{8}$$

After that, the overall correlation score is defined as the weighted sum of all source task scores, i.e., $\tau(Q_\delta, Q) = \sum_i w_i \tau_i(Q_\delta, Q)$.

**Greedy solution**. We observe that Problem 7 has transformed into a combinatorial optimization problem. We solve it approximately with a greedy algorithm. It starts with an empty set and iteratively adds the query that maximizes the correlation score, until the average cost of $Q_\delta$ exceeds $\delta$ fraction of the full query set. Algorithm 1 presents the pseudocode of the algorithm.

## 7.2 Efficient Candidate Generation

Each HB-based MFO inner loop starts with a set of candidate configurations. This section describes how MFTune recommends them.

Inspired by BOHB [10], we use BO [16] for configuration recommendation, as it balances exploration and exploitation better than Hyperband's random sampling. However, BO is often inaccurate in early iterations due to sparse observations. To mitigate this 'cold-start' issue, prior works integrate surrogate models from historical tasks to guide the search [36, 66]. In the scenario of MFO, MFES [39] further builds and combines surrogates for observations at different fidelity levels. MFTune combines these two ideas by treating both historical tasks and fidelity-specific observations as source tasks. It assigns their weights using Section 5 and combines their surrogates accordingly. To avoid scale mismatch across surrogate outputs, MFTune aggregates acquisition-score rankings rather than raw scores. For sampled and mutated candidates, let $R_i(\boldsymbol{x})$ be the rank of configuration $\boldsymbol{x}$ under source-task surrogate $M_i$. The combined rank is $R(\boldsymbol{x}) = \sum_i w_i R_i(\boldsymbol{x})$, and MFTune recommends the top-$n_1$ configurations required by the MFO outer loop.

**Two-phase warm start**. To further improve the tuning efficiency, many works warm-start with configurations selected from similar tasks [33, 36, 49]. However, warm start faces an exploration-exploitation trade-off: using too few historical configurations underuses transfer knowledge, while using too many consumes the exploration budget for the target task. To address this, MFTune designs a two-phase warm-start strategy tailored to MFO. Phase 1 is similar to existing methods. MFTune selects the best configurations from the most similar historical tasks and evaluates them at full fidelity at the beginning of tuning. After this, MFTune starts MFO and transitions to the Phase 2 warm start. Specifically, it collects the remaining configurations that outperform the median in each source task, i.e., $G_i$ in Section 6.1, and forms the warm-start set $G_{\mathrm{ws}} = \bigcup_{i=1}^{K} G_i$. Configurations in $G_{\mathrm{ws}}$ are ranked by $v(\cdot)$ in Equation 3, so better configurations from more similar tasks receive higher priority. At the start of each inner loop of MFO, except loops only containing full-fidelity evaluations, MFTune recommends several configurations from $G_{\mathrm{ws}}$ and uses BO to fill the remaining positions up to $n_1$. The number of configurations drawn from $G_{\mathrm{ws}}$ is set to the number that will reach full-fidelity evaluation in each inner loop, ensuring that promising configurations do not prune each other. Compared with existing strategies, warm start in MFTune has two strengths: 1) It adopts two levels of utilization for configurations with varying priorities. 2) It balances between exploration and exploitation by making fuller use of historical observations without hindering exploration. As most candidates remain BO-generated, and weak Phase-2 warm-start configurations will be filtered out at low-fidelity before wasting excessive resources.

## 7.3 Multi-Fidelity Optimization Process

Combining the above strategies enables MFTune to perform MFO on the compressed search space. Before MFO starts, MFTune runs Algorithm 1 once to construct query subsets for different fidelity levels. The configuration generator then uses warm start and the combined surrogate to recommend candidate configurations. Finally, MFTune follows the Hyperband framework for MFO.

**Tuning across diverse historical task scenarios**. MFTune adapts to different levels of historical-data availability. We consider three scenarios. 1) *Same-query history is available*: MFTune activates fidelity partitioning, transfer learning, and search-space compression from the beginning with weighted source tasks. 2) *No historical task has the same query set*: fidelity partitioning cannot be activated immediately because Equation 8 requires query correspondence. Thus, the target workload is evaluated at full fidelity in the initial phase. Meanwhile, transfer learning and search-space compression are still enabled from the beginning, because they rely on task-level similarity and promising configuration regions rather than identical query sets. As more target observations are gathered, the transition mechanism in Section 5 determines when the current task has accumulated sufficient evidence to serve as a source task for fidelity partitioning; then MFTune constructs low-fidelity query subsets using Equation 8 (e.g., $\tau_T(Q_\delta, Q)$) and Algorithm 1, and

**Table 1: Hardware Scenarios**

| Scenario | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| **Node** | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 |
| **CPU (cores)** | 64 | 32 | 32 | 64 | 64 | 32 | 32 | 64 |
| **RAM (GB)** | 256 | 128 | 256 | 128 | 256 | 128 | 256 | 128 |

activates MFO. 3) *No historical data is available*: MFTune initially falls back to vanilla BO. As target observations accumulate, the transition mechanism jointly enables search-space compression and MFO online. To sum up, MFTune does not permanently depend on same-query historical tasks; its components are activated either from the beginning or online depending on the available evidence.

# 8 EXPERIMENTAL EVALUATION

## 8.1 Experiment Setup

**Baselines**. We compare MFTune with six baselines. — *Three approaches without transfer learning*: *1) LOCAT* [62] gradually identifies the important knobs to reduce the search space. *2) TopTune* [60], a DBMS tuning method that compresses the search space with knob-dimensional projection and alternating between categorical and continuous knobs. *3) BOHB* [10], a generic MFO method. We adapt it to Spark SQL with data-volume reduction as the low-fidelity proxy. — *Three approaches with transfer learning*: *4) Tuneful* [11] explores significant knobs and applies a multi-task GP with similar source tasks. *5) Rover* [48] weights surrogates of source tasks by similarity. *6) LOFTune* [33] warm-starts from similar workloads and uses a workload-aware simulator to fit all historical data.

**Benchmarks**. We mainly evaluate MFTune on two typical benchmarks following literature [1, 11, 33]. *1) TPC-H*, a decision support benchmark with 22 queries designed to simulate business-oriented ad-hoc queries and concurrent data modifications. *2) TPC-DS*, a comprehensive benchmark with 99 queries for evaluating decision support systems. We utilize two data sizes (100 and 600GB) for both TPC-H and TPC-DS, resulting in four benchmark/data-size combinations. In addition, we use DSB [9] for the query-shift analysis in Section 8.3.3, because it supports controlled workload generation.

**Environment**. We conducted the experiments using Spark 3.4.4 on the OpenEuler 22.03 operating system [43]. To simulate tuning and knowledge transfer across different hardware environments, experiments were conducted on eight hardware scenarios, varying in node count (2 or 3), CPU cores (32 or 64), and memory (128GB or 256GB), as shown in Table 1. Unless otherwise specified, Scenario A serves as the default hardware environment for our evaluations.

**Historical data**. We collect task meta-features and observation histories of 32 distinct tuning tasks, derived from the combination of benchmarks, data sizes, and eight hardware scenarios. Each meta-feature is a 34-d vector extracted from SparkEventLog. The history for each task was prepared by running vanilla Bayesian optimization [16] and collecting 50 observations. By default, we use a leave-one-out evaluation setting, where each target task is tuned using histories from the other 31 tasks.

**Tuning Setting.** *1) Optimization target*: In all experiments, we optimize the total latency (i.e., the sum of all query times) of the target Spark SQL workload, and use elapsed time as the cost metric for fidelity partitioning. *2) Search space*: We extend the search space in Tuneful [11] to include 60 parameters that influence the application performance. More details are available in the code repository. *3) Tuning budget*: Except in the cold-start setting, each method runs for 48 hours per target task, and we report the best performance among the configurations it recommends and evaluates. *4) Implementation details*: MFTune implements BO based on OpenBox [19], a toolkit for black-box optimization. We set the cumulative density threshold $\alpha = 0.65$. For MFO, we set the reduction factor $\eta = 3$ and the maximum resource $R = 9$ following Section 3.3. This setting yields two low-fidelity levels, 1/9 and 1/3 of the full resource.

## 8.2 Main Results under the Default Setting

This section evaluates MFTune on TPC-H and TPC-DS with Hardware Scenario A and 600GB data. We adopt a leave-one-out pattern, where each target task uses histories from the other 31 tasks. Since the repository includes source tasks with the same query set as the target, MFTune can leverage historical knowledge to activate fidelity partitioning and MFO from the beginning.

The experimental results are illustrated in Figure 3a and 3d, from which we can draw the following conclusions: 1) methods that leverage historical data (MFTune, Tuneful, Rover, and LOFTune) generally exhibit superior performance and faster convergence compared to those starting from scratch (LOCAT, TopTune, and BOHB), demonstrating that historical knowledge from similar tasks can significantly accelerates the search for high-quality Spark SQL configurations. Although BOHB uses multi-fidelity evaluation, its data-volume proxy provides less reliable full-workload feedback for Spark SQL, making its performance close to other no-history baselines. 2) Among the approaches utilizing historical knowledge, MFTune shows the strongest convergence. While LOFTune initially achieves rapid latency reduction through warm start, its use of history is limited to initialization. This limited use of history leads to stagnation, allowing Rover and Tuneful to eventually surpass it. Tuneful converges faster than Rover in the early stage because it combines transfer learning with search-space compression, while Rover does not reduce the search space. Although these baselines leverage transfer learning to accelerate convergence, their exclusive reliance on full-fidelity evaluations inevitably leads to time-consuming explorations of poor configurations. In contrast, MFTune employs MFO to filter poor configurations at low fidelity, and further strengthens this process with search-space compression and tailored transfer learning, enabling sustained latency reduction throughout tuning. 3) MFTune yields the best final tuning results across both benchmarks. Specifically, compared to the six baselines, MFTune reduces latency of Spark SQL ranging from 25.9% to 43.1% on TPC-H and 37.8% to 69.3% on TPC-DS, respectively.

## 8.3 Evaluation on Generalization

This subsection evaluates the generalization of MFTune in four settings: cross-benchmark transfer, cold start, transfers with varying query overlap, and transfers across data sizes and hardwares.

*8.3.1 Results in Cross-Benchmark Transfer.* This section evaluates MFTune's robustness under cross-benchmark transfer, where source and target workloads come from different benchmarks. We compare MFTune against the three baselines that utilize historical data

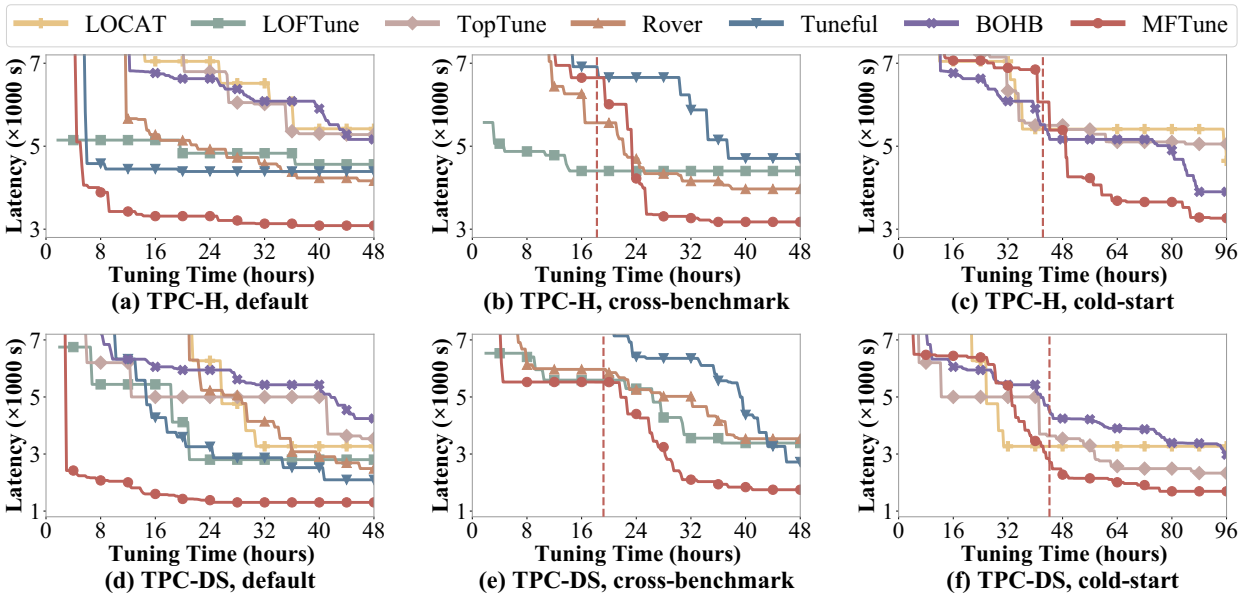



**Figure 3: Tuning performance over time across six scenarios. Each curve shows the average best-observed latency over three runs. The vertical red dashed line marks when MFTune activates MFO if fidelity partitioning is initially inapplicable.**

(Tuneful, Rover, and LOFTune), which are the top-3 baselines in the default setting. In the cross-benchmark setting, when TPC-H is the target benchmark, the framework uses 16 source tasks only from TPC-DS, and vice versa. As discussed in Section 7.3, without same-query history, MFTune cannot activate fidelity partitioning at the beginning. Thus, MFTune initially uses transfer learning and search-space compression with full-fidelity evaluations. Once sufficient target observations are collected, it dynamically constructs low-fidelity query subsets and activates MFO.

The experimental results are shown in Figure 3b and 3e. The vertical red dashed line marks the specific time point MFTune enables MFO once fidelity partitioning becomes applicable with sufficient observations. Following this activation, the performance gap between MFTune and the baselines widens significantly as low-fidelity proxies enable high-frequency explorations beyond the reach of full-fidelity methods. Overall, MFTune also achieves the best tuning results under cross-benchmark transfer. Compared with the baselines, MFTune achieves a latency reduction ranging from 20.0% to 32.5% on TPC-H and 35.7% to 50.6% on TPC-DS, showing its adaptability when same-query history is unavailable.

*8.3.2 Results without Historical Data.* To further evaluate the robustness of MFTune in extreme scenarios, we conduct experiments under a cold-start setting where no historical data is available. In this subsection, we compare MFTune against TopTune, LOCAT, and BOHB, the three baselines that do not rely on historical knowledge. Unlike the previous settings, the tuning budget here is extended to 96 hours to allow for sufficient exploration from scratch on the 600GB data scale. As detailed in Section 7.3, since there are no historical tasks to leverage, MFTune initially degrades to a vanilla BO. As the process continues and observations accumulate, the system automatically enables space compression and MFO.

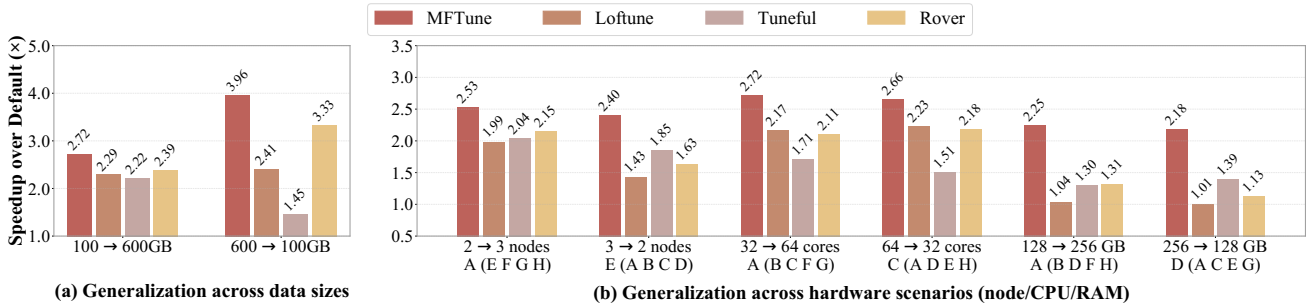
Figure 3c and 3f illustrate the experimental results for TPC-H and TPC-DS, respectively. During the initial tuning phase, MFTune exhibits a similar convergence trend to the baselines while gathering the necessary observations. However, once MFO is activated (marked by the vertical red dashed line), the convergence speed increases significantly, demonstrating that MFTune can successfully identify and utilize low-fidelity proxies online even without prior history. Eventually, MFTune achieves a much lower final latency compared to TopTune and LOCAT, which remain limited by the high temporal cost of performing exclusive full-fidelity evaluations. Specifically, MFTune outperforms all no-history baselines by 16.3% to 35.4% on TPC-H, and by 27.4% to 48.2% on TPC-DS.

*8.3.3 Results under Varying Degrees of Query Shift.* The default and cross-benchmark settings represent two endpoints of history availability: the former has same-query history, while the latter has no overlap between source and target query sets. To further examine MFTune when no historical task has the same query set, we construct controlled workload shift with varying degrees of query overlap using DSB [9]. We choose DSB for two reasons: first, it enhances the data distributions of TPC-DS and provides richer query templates; second, it provides explicit query-type classifications for templates, which allows us to construct semantically meaningful workload shifts rather than arbitrary splits. DSB queries are divided into three groups: A contains 15 aggregation queries, B contains 22 multi-block queries, and C contains 15 Select-Project-Join queries. We use AB as the source workload, which contains 37 queries, and evaluate three target workloads with decreasing overlap: ABC (52 queries), AC (30 queries), and C (15 queries).

Table 2 reports the gains of using AB history over no history under different query-overlap levels. Although MFO cannot be activated initially, AB history still improves early tuning by guiding

Table 2: Effect of query overlap on DSB. For example, AB→AC means tuning AC with AB as the source. Latencies are reported in $10^3$ seconds. Early best is measured at 25% of the tuning budget. Gains are computed relative to MFTune without AB history.

| Workload shift | Overlap | Default latency | Early best latency | | | Final best latency | | |
|---|---|---|---|---|---|---|---|---|
| | | | w/o AB | w/ AB | Gain | w/o AB | w/ AB | Gain |
| AB→ABC | 71.2% | 3.22 | 2.79 | 0.98 | 64.9% | 1.58 | 0.82 | 48.1% |
| AB→AC | 50.0% | 1.86 | 1.42 | 1.01 | 28.9% | 0.71 | 0.48 | 32.7% |
| AB→C | 0.0% | 0.84 | 0.62 | 0.58 | 6.5% | 0.26 | 0.22 | 15.4% |




(a) Generalization across data sizes

(b) Generalization across hardware scenarios (node/CPU/RAM)

Figure 4: Generalization performance of transfer-learning-based tuning methods across diverse settings. Results represent the speedup of the optimal latency achieved within 48 hours relative to the default Spark configuration.

transfer learning and search-space compression toward promising configurations and knob regions. As overlap decreases from 71.2% to 50.0% and 0.0%, the early gain drops from 64.9% to 28.9% and 6.5%, indicating weaker immediate guidance from less-similar histories. The final gain additionally includes the effect of MFO after target observations accumulate, reaching 48.1%, 32.7%, and 15.4% in the three settings. These results show a general consistency between query overlap and transfer benefits, indicating that MFTune can exploit more useful historical information when source tasks are more related. However, consistent with the cross-benchmark transfer, even under complete query drift, historical transfer plus delayed MFO outperforms tuning without history.

*8.3.4 Results with Varying Data Size and Hardware.* This subsection evaluates MFTune on TPC-H under data-size and hardware shifts. Figure 4a reports cross-scale transfer between 100GB and 600GB on Hardware A. When 600GB is the target, the 16 tasks from the 100GB setting are source tasks, and vice versa. Furthermore, Figure 4b reports transfer across different hardware configurations, including node count, CPU cores, and memory. For example, in the 2 → 3 node setting, Hardware A (600GB) is the target, and the 16 tasks from 2-node environments (Hardware E, F, G, H) are source tasks.

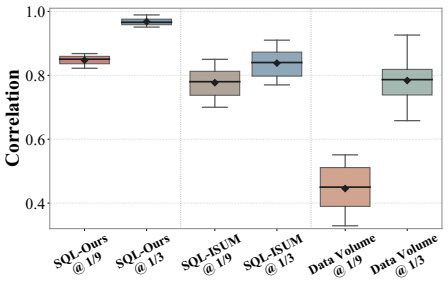
The experimental results demonstrate the superior robustness of MFTune compared to existing transfer-learning-based methods. MFTune consistently achieves the highest latency speedup across all tested scenarios. Notably, it reaches a peak speedup of 3.96x in the 600 → 100 GB data size transfer. In hardware-shift scenarios, while baselines exhibit significant performance volatility, MFTune maintains robust speedups exceeding 2.18x. These results show that across different historical transfer scenarios, MFTune can adaptively identify and weight similar source tasks through similarity identification and exploit their information to accelerate tuning.

## 8.4 Detailed Analysis

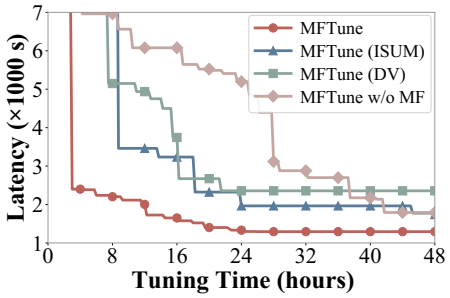
*8.4.1 Analysis of Multi-Fidelity Optimization.* This subsection analyzes the effectiveness of MFTune's MFO mechanism on TPC-DS.

We first examine whether existing DBMS workload-compression methods can construct query subsets for Spark SQL MFO. These methods cannot be directly applied mainly because their original metrics target DBMS downstream tasks such as index selection and Approximate Query Processing (AQP). They rely on signals or tools such as candidate indexes, hypothetical-index what-if interfaces, and AQP-specific error models, which are unavailable in our Spark SQL tuning task. Therefore, we keep their subset-selection algorithms where possible, but replace their original signals with transfer-based Spark SQL adaptations. 1) SQL-Dist [6] uses application-specific query distances for index selection or AQP. Since these distances cannot be directly used in Spark SQL tuning, we redefine query distance as $1-\tau$, where $\tau$ is the Kendall correlation between two queries' source-task latency vectors. 2) SQL-GSUM [8] requires SQL text and execution statistics to preserve feature coverage. We adapt the latter one with SparkEventLog. 3) SQL-ISUM [50] defines utility and influence for index-tuning improvement. We adapt the utility by source-task latency improvement over the default configuration and similarity by Kendall correlation between source-task latency vectors. Overall, these adaptations favor the DBMS workload-compression baselines: SQL-Dist and SQL-ISUM are strengthened with source-task transfer signals from MFTune, despite being originally inapplicable to Spark SQL tuning.

Figure 1b compares the fidelity quality of these methods across different fidelity levels. SQL-ISUM and SQL-Dist outperform SQL-GSUM because our adaptations strengthen their utility/similarity or distance metrics with source-task latency correlations. However, they still underperform SQL-Ours, showing that the mismatch is




(a) Fidelity correlation.




(b) Performance over time.

| System telemetry | 1/9 fidelity | 1/3 fidelity |
|---|---|---|
| Avg peak memory | 0.86 | 0.92 |
| GC time | 0.43 | 0.57 |
| Shuffle read | 0.73 | 0.87 |
| Spill rate | 0.64 | 0.57 |

(c) Correlation on system hidden states.

**Figure 5: Effectiveness of the MFO mechanism on TPC-DS.**

not only the required signals. Specifically, their subset-selection algorithms remain misaligned with MFO. MFTune directly optimizes whether the subset preserves the full-workload configuration ranking, while these methods indirectly encourage representativeness through query-to-query distance or influence. This direct-versus-indirect optimization gap explains why adapted workload-compression baselines still fall behind SQL-Ours. Overall, SQL-Ours performs better for two reasons: 1) its metric targets Spark SQL tuning feedback, and 2) its greedy selection directly optimizes the consistency with the full-workload required by MFO. Figure 5a further tests this conclusion across 16 TPC-DS workloads. SQL-Ours achieves the highest and most concentrated correlations, with means of 0.847 at 1/9 and 0.968 at 1/3, outperforming both SQL-ISUM (0.777 and 0.838) and Data Volume (0.445 and 0.784).

We further analyze why the selected low-fidelity subset correlates well with full-workload latency. Figure 5c reports the Kendall correlation between low-fidelity and full-fidelity telemetry across 50 random configurations. We examine four representative Spark SQL hidden-state indicators: peak execution memory, JVM GC time, shuffle read bytes, and spill rate normalized by running time. The low-fidelity subset preserves the peak memory and shuffle signals well: correlations reach 0.86/0.92 for peak memory and 0.73/0.87 for shuffle read at 1/9/1/3 fidelity. GC time and spill rate are noisier: GC is sensitive to JVM/runtime state, while spill rate can be dominated by a few memory-intensive operators or skewed partitions. Nevertheless, they still show non-trivial correlation. These results provide system-level evidence that the subset preserves key bottlenecks, supporting its high correlation with full-workload latency.

We finally evaluate whether the fidelity-quality advantage in Figure 1b improves end-to-end tuning. Specifically, we compare MFTune with three variants: 1) MFTune w/o MF, which relies only on full-fidelity evaluations; 2) MFTune (DV), which uses data-volume reduction as the low-fidelity proxy; and 3) MFTune (ISUM), which replaces our SQL Selection with adapted SQL-ISUM. We choose SQL-ISUM because it has the best fidelity correlation among adapted DBMS workload-compression baselines in Figure 1b. Figure 5b shows that MFTune converges faster and achieves a better final configuration than all variants. Compared with MFTune w/o MF, MFTune (DV), and MFTune (ISUM), MFTune reduces latency by 27.8%, 45.1%, and 25.6%, respectively. This shows that other fidelity partitioning strategies are insufficient for Spark SQL MFO.

*8.4.2 Analysis of Space Compression.* In this section, we evaluate the effectiveness of the search space compression (SC) mechanism in MFTune and analyze the sensitivity of hyperparameter $\alpha$.

We first compare MFTune 's space compression with the following baselines: *1) w/o SC*, the uncompressed search space. *2) Box* [45] suggests a smallest box covering the best-observed configurations of source tasks. *3) Decrease*, adopted in Tuneful [11], which removes 40% unimportant knobs every 10 iterations based on the importance ranking. *4) Project*, adopted in LlamaTune [23] and TopTune [60], which projects the original space to a synthetic low-dimensional one and narrows ranges by bucketization. *5) Vote*, adopted in OpAdvisor [68], which extracts boundary values of discrete historical configurations and votes to generate the target space. Specifically, we replace MFTune's SC component with each baseline strategy and run end-to-end comparisons on TPC-H 600GB.

As shown in Figure 6a, MFTune maintains an advantage throughout tuning, with a final latency reduction of 14.8%–35.7% over the variants. Existing strategies show clear limitations: Decrease relies on target observations, which are often insufficient for accurate knob ranking; Box can over-prune by ignoring task-specific differences; and Project reduces dimensionality but cannot exclude poor subspaces precisely. Vote is the closest to our approach, its reliance on discrete boundaries makes it highly sensitive to outliers. In contrast, MFTune employs Kernel Density Estimation (KDE) to smooth out noise and identify robust high-potential regions. To isolate the effect of SC, we further conduct a stress test by disabling the warm start. The rationale is that warm start can hide the weakness of a suboptimal search space by directly evaluating strong historical configurations at the beginning. As shown in Figure 6b, without warm start, the gap between MFTune and the variants becomes larger, and MFTune reduces latency by 20.4%–43.0%. This demonstrates the robustness of our density-based SC strategy.

Finally, Figure 6c analyzes the sensitivity of the cumulative density threshold $\alpha$. This threshold controls the trade-off between pruning strength and conservativeness. When $\alpha$ is too small, e.g., 0.5, the space is over-pruned and may exclude the optimum, which causes early stagnation. When $\alpha$ is too large, e.g., 0.8, compression is too conservative and retains too many low-potential regions, which may slow the convergence. Between these extremes, MFTune is stable for $\alpha \in [0.6, 0.7]$, achieving comparable convergence. Therefore, we set $\alpha = 0.65$ by default.

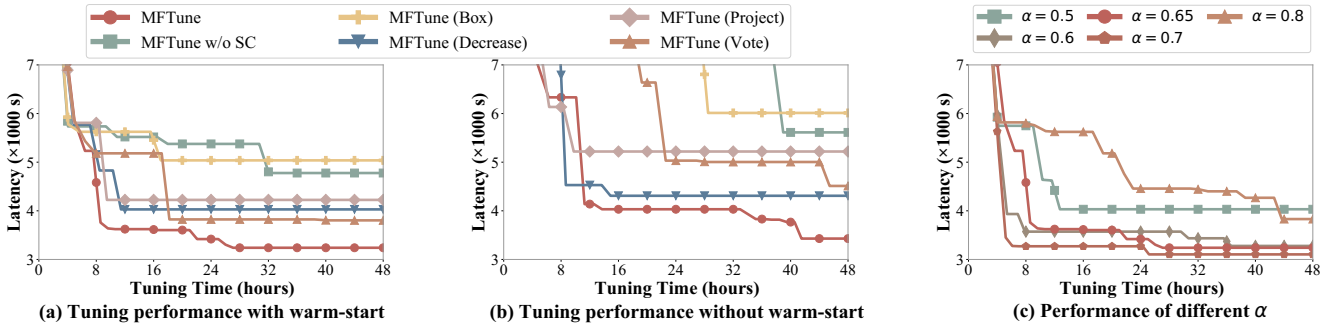



**Figure 6: Analysis of the search space compression (SC) mechanism on TPC-H.**

**Table 3: Ablation study of warm-start phases on TPC-H (600GB). P1 and P2 denote Phase 1 warm start and Phase 2 warm start, respectively. All metrics represent the relative gains achieved by MFTune compared to each specific variant.**

| P1 | P2 | Latency Reduction (%) | Tuning Acceleration |
|---|---|---|---|
| × | × | 5.50% | 2.15× |
| ✓ | × | 5.13% | 1.98× |
| × | ✓ | 1.25% | 1.12× |

**Table 4: Noise study on TPC-H (600GB). Best retained $a \rightarrow b$ denotes the ratio that the clean best configuration at fidelity $a$ is still promoted to fidelity $b$ after noise injection.**

| Noise level | Latency change vs. clean | Best retained 1/9→1/3 | Best retained 1/3→1 |
|---|---|---|---|
| $\sigma = 0.05$ | -1.23% | 95.0% | 78.5% |
| $\sigma = 0.10$ | 3.17% | 90.5% | 70.0% |
| $\sigma = 0.15$ | 9.64% | 82.4% | 55.2% |

*8.4.3 Ablation Study of Warm Start.* This subsection evaluates the contribution of the two-phase warm-start strategy. As shown in Table 3, the two-phase design outperforms both single-phase variants. Phase 1 provides a strong initial configuration, while Phase 2 drives further improvement by working with MFO to filter and exploit more candidates at low fidelity. Overall, the two-phase strategy makes full use of historical data and balances fast initialization with sustained exploration, achieving a 5.50% latency reduction and 2.15× tuning acceleration over the cold-start baseline.

*8.4.4 Robustness to Observational Noise.* In real clusters, Spark SQL evaluations can be affected by transient noise from network jitter, CPU contention, or other background activities. Thus, we conduct a controlled noise-injection study to test the stability of MFTune under increasingly severe noise. Specifically, for each evaluated latency $y$, we inject multiplicative noise as $\tilde{y} = y \cdot \max(1 + \epsilon, 0.01)$, where $\epsilon \sim \mathcal{N}(0, \sigma^2)$. We consider three noise levels: $\sigma = 0.05$, 0.10, and 0.15. Table 4 shows that MFTune remains stable under increasing observational noise. 1) The 1/9→1/3 promotion stage has high retention rates, remaining above 82.4% under the three noise levels. This is because this stage promotes multiple configurations, e.g., 9 to 3, giving the best configuration room to survive noisy perturbations. 2) In contrast, the 1/3→1 stage is stricter, often promoting only one configuration from 3 or 5 candidates. Therefore, its retention rates drop to 78.5%, 70.0%, and 55.2%. 3) However, lower second-stage retention does not translate into severe tuning degradation. Even under the highest noise, latency increases by only 9.64%, still smaller than the gaps between MFTune and competing baselines in the main experiments. This is because configurations reaching 1/3 fidelity are already promising, so selecting a near-best candidate can still yield competitive final performance.

*8.4.5 Analysis of Tuning Overhead.* We record the computational overhead of MFTune under the default settings. For task-level initialization, source-task similarity prediction takes ~15s. Fidelity partitioning takes ~21s for TPC-DS compared to ~0.5s for TPC-H, mainly because TPC-DS has many more queries. At the iteration level, similarity calculation via Kendall takes ~0.6s, search-space compression takes ~2s, and BO candidate recommendation takes ~0.2s. Within search-space compression, SHAP computation takes ~0.3s on average. Collectively, these overheads are negligible compared with configuration evaluations, which take thousands of minutes, and total tuning budgets of dozens of hours.

## 9 CONCLUSION

In this paper, we propose MFTune, a multi-fidelity framework that overcomes the prohibitive costs and high dimensionality of Spark SQL tuning through query-based fidelity partitioning and density-based search space compression. By integrating a two-phase warm start and adapted transfer learning, MFTune achieves a superior balance between historical knowledge reuse and efficient exploration. Experimental results on TPC-H and TPC-DS benchmarks confirm that MFTune consistently delivers optimal performance across all evaluated settings, significantly outperforming five state-of-the-art methods with negligible computational overhead.

## ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (U23B2048, U22B2037), the Beijing Municipal Natural Science Foundation (Grant No. L2603008), and the "Fundamental Research Funds for the Central Universities, Peking University". Bin Cui is the corresponding author.

# REFERENCES


[1] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. {CherryPick}: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 469–482.

[2] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1383–1394.

[3] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*. 2546–2554.

[4] Baoqing Cai, Yu Liu, Ce Zhang, Guangyu Zhang, Ke Zhou, Li Liu, Chunhua Li, Bin Cheng, Jie Yang, and Jiashu Xing. 2022. HUNTER: an online cloud database hybrid tuning system for personalized requirements. In *Proceedings of the 2022 International Conference on Management of Data*. 646–659.

[5] Zhuo Chang, Xinyi Zhang, Yang Li, Xupeng Miao, Yanzhao Qin, and Bin Cui. 2024. Mfix: An efficient and reliable index advisor via multi-fidelity bayesian optimization. In *2024 IEEE 40th international conference on data engineering (ICDE)*. IEEE, 4343–4356.

[6] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek Narasayya. 2002. Compressing sql workloads. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 488–499.

[7] Chao Chen, Jinhan Xin, and Zhibin Yu. 2024. TIE: Fast experiment-driven ml-based configuration tuning for in-memory data analytics. *IEEE Trans. Comput.* 73, 5 (2024), 1233–1247.

[8] Shaleen Deep, Anja Gruenheid, Paraschos Koutris, Jeffrey Naughton, and Stratis Viglas. 2020. Comprehensive and efficient workload compression. *Proceedings of the VLDB Endowment* 14, 3 (2020), 418–430.

[9] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek R. Narasayya. 2021. DSB: A decision support benchmark for workload-driven and traditional database systems. *Proceedings of the VLDB Endowment* 14, 13 (2021), 3376–3388.

[10] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and efficient hyperparameter optimization at scale. In *International conference on machine learning*. PMLR, 1437–1446.

[11] Ayat Fekry, Lucian Carata, Thomas Pasquier, Andrew Rice, and Andy Hopper. 2020. To tune or not to tune? in search of optimal configurations for data analytics. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2494–2504.

[12] Paolo Frazzetto, Muhammad Uzair Ul Haq, Flavia Fabris, and Alessandro Sperduti. 2025. Graph Neural Networks for Candidate-Job Matching: An Inductive Learning Approach: P. Frazzetto et al. *Data Science and Engineering* (2025), 1–18.

[13] 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*. 954–973.

[14] Yi-Qi Hu, Yang Yu, Wei-Wei Tu, Qiang Yang, Yuqiang Chen, and Wenyuan Dai. 2019. Multi-fidelity automatic hyper-parameter tuning via transfer series expansion. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 3846–3853.

[15] Shiyue Huang, Yanzhao Qin, Xinyi Zhang, Yaofeng Tu, Zhongliang Li, and Bin Cui. 2023. Survey on performance optimization for database systems. *Science China Information Sciences* 66, 2 (2023), 121102.

[16] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*. Springer, 507–523.

[17] Chip Huyen. 2022. *Designing machine learning systems.* " O'Reilly Media, Inc.".

[18] Kevin Jamieson and Ameet Talwalkar. 2016. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial intelligence and statistics*. PMLR, 240–248.

[19] Huaijun Jiang, Yu Shen, Yang Li, Beicheng Xu, Sixian Du, Wentao Zhang, Ce Zhang, and Bin Cui. 2024. Openbox: A Python toolkit for generalized black-box optimization. *Journal of Machine Learning Research* 25, 120 (2024), 1–11.

[20] Jiantong Jiang, Zeyi Wen, Atif Mansoor, and Ajmal Mian. 2024. Efficient hyperparameter optimization with adaptive fidelity identification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 26181–26190.

[21] Jiaqi Jin, Ziquan Fang, Lu Chen, and Yunjun Gao. 2025. PostMan: A Productive System for Spatio-temporal Data Management and Analysis: J. Jin et al. *Data Science and Engineering* (2025), 1–24.

[22] Donald R Jones, Matthias Schonlau, and William J Welch. 1998. Efficient global optimization of expensive black-box functions. *Journal of Global optimization* 13, 4 (1998), 455–492.

[23] 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.

[24] Ralph Kimball and Margy Ross. 2013. *The data warehouse toolkit: The definitive guide to dimensional modeling*. John Wiley & Sons.

[25] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. 2017. Fast bayesian optimization of machine learning hyperparameters on large datasets. In *Artificial intelligence and statistics*. PMLR, 528–536.

[26] Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. 2017. Learning curve prediction with Bayesian neural networks. In *International conference on learning representations*.

[27] Aaron Klein, Louis C Tiao, Thibaut Lienart, Cedric Archambeau, and Matthias Seeger. 2020. Model-based asynchronous hyperparameter and neural architecture search. *arXiv preprint arXiv:2003.10865* (2020).

[28] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2025. GPTuner: An LLM-Based Database Tuning System. *ACM SIGMOD Record* 54, 1 (2025), 101–110.

[29] Ben Letham, Roberto Calandra, Akshara Rai, and Eytan Bakshy. 2020. Re-examining linear embeddings for high-dimensional Bayesian optimization. *Advances in neural information processing systems* 33 (2020), 1546–1558.

[30] 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.

[31] Guo-Liang Li, Jiang Wang, and Guo Chen. 2024. openGauss: An enterprise-grade open-source database system. *Journal of Computer Science and Technology* 39, 5 (2024), 1007–1028.

[32] Jin Li, Quan Chen, Xiao-Xin Tang, and Min-Yi Guo. 2024. Sha: Qos-aware software and hardware auto-tuning for database systems. *Journal of Computer Science and Technology* 39, 2 (2024), 369–383.

[33] Jiahui Li, Junhao Ye, Yuren Mao, Yunjun Gao, and Lu Chen. 2025. LOFTune: A Low-Overhead and Flexible Approach for Spark SQL Configuration Tuning. *IEEE Transactions on Knowledge and Data Engineering* (2025).

[34] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2018. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research* 18, 185 (2018), 1–52.

[35] Lisha Li, Kevin Jamieson, Afshin Rostamizadeh, Katya Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. 2018. Massively parallel hyperparameter tuning. (2018).

[36] Yang Li, Huaijun Jiang, Yu Shen, Yide Fang, Xiaofeng Yang, Danqing Huang, Xinyi Zhang, Wentao Zhang, Ce Zhang, Peng Chen, et al. 2023. Towards General and Efficient Online Tuning for Spark. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3570–3583.

[37] Yiyan Li, Haoyang Li, Jing Zhang, Renata Borovica-Gajic, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, Cuiping Li, and Hong Chen. 2025. AgentTune: An Agent-Based Large Language Model Framework for Database Knob Tuning. *Proceedings of the ACM on Management of Data* 3, 6 (2025), 1–29.

[38] Yang Li, Yu Shen, Huaijun Jiang, Wentao Zhang, Jixiang Li, Ji Liu, Ce Zhang, and Bin Cui. 2022. Hyper-tune: towards efficient hyper-parameter tuning at scale. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1256–1265.

[39] Yang Li, Yu Shen, Jiawei Jiang, Jinyang Gao, Ce Zhang, and Bin Cui. 2021. Mfes-hb: Efficient hyperband with multi-fidelity quality measurements. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 8491–8500.

[40] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. *Advances in neural information processing systems* 30 (2017).

[41] Felix Mohr and Jan N van Rijn. 2024. Learning curves for decision making in supervised machine learning: a survey. *Machine Learning* 113, 11 (2024), 8371–8425.

[42] Amin Nayebi, Alexander Munteanu, and Matthias Poloczek. 2019. A framework for Bayesian optimization in embedded subspaces. In *International Conference on Machine Learning*. PMLR, 4752–4761.

[43] OpenEuler. 2025. *An Open Source OS for Digital Infrastructure.* https://www.openeuler.org/ Last accessed: July 5, 2026.

[44] Emanuel Parzen. 1962. On estimation of a probability density function and mode. *The annals of mathematical statistics* 33, 3 (1962), 1065–1076.

[45] Valerio Perrone, Huibin Shen, Matthias W Seeger, Cedric Archambeau, and Rodolphe Jenatton. 2019. Learning search spaces for bayesian optimization: Another view of hyperparameter transfer learning. *Advances in neural information processing systems* 32 (2019).

[46] David Salinas, Jacek Golebiowski, Aaron Klein, Matthias Seeger, and Cedric Archambeau. 2023. Optimizing hyperparameters with conformal quantile regression. In *International Conference on Machine Learning*. PMLR, 29876–29893.

[47] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. 2019. Presto: SQL on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1813.

[48] 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 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4800–4812.

[49] Yu Shen, Beicheng Xu, Yupeng Lu, Donghui Chen, Huaijun Jiang, Zhipeng Xie, Senbo Fu, Nan Zhang, Yuxin Ren, Ning Jia, et al. 2025. A-Tune-Online: Efficient and QoS-Aware Online Configuration Tuning for Dynamic Workloads. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 2161–2173.

[50] Tarique Siddiqui, Saehan Jo, Wentao Wu, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2022. ISUM: Efficiently compressing large and complex workloads for scalable index tuning. In *Proceedings of the 2022 International Conference on Management of Data*. 660–673.

[51] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*. 2951–2959.

[52] SparkConf. 2025. *Configuration - Spark 4.0.1 Documentation*. Stanford University. https://spark.apache.org/docs/latest/configuration.html Last accessed: July 5, 2026.

[53] Wenwen Sun, Zhicheng Pan, Zirui Hu, Yu Liu, Chengcheng Yang, Rong Zhang, and Xuan Zhou. 2025. Rabbit: Retrieval-Augmented Generation Enables Better Automatic Database Knob Tuning. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 3807–3820.

[54] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1626–1629.

[55] Robert Tibshirani. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society Series B: Statistical Methodology* 58, 1 (1996), 267–288.

[56] Immanuel Trummer. 2022. DB-BERT: a Database Tuning Tool that" Reads the Manual". In *Proceedings of the 2022 international conference on management of data*. 190–203.

[57] 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 2017 ACM international conference on management of data*. 1009–1024.

[58] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2022. UDO: Universal Database Optimization using Reinforcement Learning. In *Proceedings of the VLDB Endowment*, Vol. 14. 3402–3414.

[59] James Warren and Nathan Marz. 2015. *Big Data: Principles and best practices of scalable realtime data systems*. Simon and Schuster.

[60] Rukai Wei, Yu Liu, Yufeng Hou, Heng Cui, Yongqiang Zhang, and Ke Zhou. 2025. TopTune: Tailored Optimization for Categorical and Continuous Knobs Towards Accelerated and Improved Database Performance Tuning. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 613–626.

[61] Martin Wistuba and Tejaswini Pedapati. 2020. Learning to rank learning curves. In *International Conference on Machine Learning*. PMLR, 10303–10312.

[62] Jinhan Xin, Kai Hwang, and Zhibin Yu. 2022. Locat: Low-overhead online configuration auto-tuning of spark sql applications. In *Proceedings of the 2022 International Conference on Management of Data*. 674–684.

[63] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *2nd USENIX workshop on hot topics in cloud computing (HotCloud 10)*.

[64] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 international conference on management of data*. 415–432.

[65] 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.

[66] 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 2021 international conference on management of data*. 2102–2114.

[67] Xinyi Zhang, Hong Wu, Yang Li, Jian Tan, Feifei Li, and Bin Cui. 2022. Towards dynamic and safe configuration tuning for cloud databases. In *Proceedings of the 2022 International Conference on Management of Data*. 631–645.

[68] Xinyi Zhang, Hong Wu, Yang Li, Zhengju Tang, Jian Tan, Feifei Li, and Bin Cui. 2023. An efficient transfer learning based configuration adviser for database tuning. *Proceedings of the VLDB Endowment* 17, 3 (2023), 539–552.

[69] Xinyang Zhao, Xuanhe Zhou, and Guoliang Li. 2023. Automatic database knob tuning: A survey. *IEEE Transactions on Knowledge and Data Engineering* 35, 12 (2023), 12470–12490.

[70] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A learning-to-rank query optimizer. *arXiv preprint arXiv:2302.06873* (2023).

[71] Yiwen Zhu, Rathijit Sen, Brian Kroth, Sergiy Matusevych, Andreas Mueller, Tengfei Huang, Rahul Challapalli, Weihan Tang, Xin He, Mo Liu, et al. 2025. Rockhopper: A robust optimizer for spark configuration tuning in production environment. In *Companion of the 2025 International Conference on Management of Data*. 743–756.