An Efficient Transfer Learning Based Configuration Adviser for Database Tuning

Xinyi Zhang*
Peking University
zhang_xinyi@pku.edu.cn

Hong Wu†
Alibaba Group
hong.wu@alibaba-inc.com

Yang Li*
Peking University
liyang.cs@pku.edu.cn

Zhengju Tang*
Peking University
tzj_jwlfp_fxz@stu.pku.edu.cn

Jian Tan†
Alibaba Group
j.tan@alibaba-inc.com

Feifei Li†
Alibaba Group
lifeifei@alibaba-inc.com

Bin Cui*‡
Peking University
bin.cui@pku.edu.cn

ABSTRACT

In recent years, a wide spectrum of database tuning systems have emerged to automatically optimize database performance. However, these systems require a significant number of workload runs to deliver a satisfactory level of database performance, which is time-consuming and resource-intensive. While many attempts have been made to address this issue by using advanced search optimizers, empirical studies have shown that no single optimizer can dominate the rest across tuning tasks with different characteristics. Choosing an inferior optimizer may significantly increase the tuning cost. Unfortunately, current practices typically adopt a single optimizer or follow simple heuristics without considering the task characteristics. Consequently, they fail to choose the most suitable optimizer for a specific task. Furthermore, constructing a compact search space can significantly improve the tuning efficiency. However, current practices neglect the setting of the value range for each knob and rely on a large number of workload runs to select important knobs, resulting in a considerable amount of unnecessary exploration in ineffective regions.

To pursue efficient database tuning, in this paper, we argue that it is imperative to have an approach that can judiciously determine a precise space and search optimizer for an arbitrary tuning task. To this end, we propose OpAdviser, which exploits the information learned from historical tuning tasks to guide the search space construction and search optimizer selection. Our design can greatly accelerate the tuning process and further reduce the required workload runs. Given a tuning task, OpAdviser learns the geometries of search space, including important knobs and their effective regions, from relevant previous tasks. It then constructs the target search space from the geometries according to the on-the-fly task similarity, which allows for adaptive adjustment of the target space. OpAdviser also employs a pairwise ranking model to capture the relationship from task characteristics to optimizer rankings. This ranking model is invoked during tuning and predicts the best optimizer to be used for the current iteration. We conduct extensive evaluations across a diverse set of workloads, where OpAdviser achieves 9.2% higher throughput and significantly reduces the number of workload runs with an average speedup of ~3.4× compared to state-of-the-art tuning systems.

PVLDB Reference Format:
Xinyi Zhang, Hong Wu, Yang Li, Zhengju Tang, Jian Tan, Feifei Li, and Bin Cui. An Efficient Transfer Learning Based Configuration Adviser for Database Tuning. PVLDB, 17(3): 539 - 552, 2023.
doi:10.14778/3632093.3632114

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

1 INTRODUCTION

Optimizing the configuration of a database management system (DBMS) is a critical aspect to achieve high system performance. Conventionally, the tuning task has been performed manually by database administrators (DBAs), involving workload and hardware configuration analysis, selection of appropriate configurations, and performance testing. However, the manual tuning process is time-consuming and expertise-consuming when dealing with complex modern workloads and various hardware environments, especially in the cloud environment. Consequently, automatic configuration tuning has attracted lots of attention in the research community [4, 11, 16, 26, 32, 33, 37, 42–44, 46, 47, 52, 54].

The current systems for configuration tuning share a generic workflow. Initially, the tuning system defines a *search space* that encompasses all possible configurations, given a target workload and a performance metric. Subsequently, the system iteratively explores configurations within the *search space* according to the suggestion from a *search optimizer*, which aims to find the configuration that maximizes the database performance. Notably, evaluating a configuration necessitates resources and time to run the workload, which dominates as the major cost when tuning databases. Minimizing the number of workload runs for finding a good configuration is a crucial requirement to adopt the tuning systems in practical scenarios, such as production services with numerous databases [24]. A sophisticated tuning system should deliver a satisfactory level of

*School of CS & Key Laboratory of High Confidence Software Technologies, Peking University
†Database and Storage Laboratory, Damo Academy, Alibaba Group
‡National Engineering Laboratory for Big Data Analysis and Applications, Peking University

Proceedings of the VLDB Endowment, Vol. 17, No. 3 ISSN 2150-8097.
doi:10.14778/3632093.3632114

database performance with minimal cost of workload runs. Given a tuning task, the construction of *search space* and the choice of *search optimizer* are the main factors that affect the tuning efficiency. While prior research has tried to enhance the tuning efficiency by selecting important knobs [5, 17, 23] and developing advanced *search optimizers* [22, 24, 50, 53, 56], when applying these tuning systems in practice, we find the following issues and challenges.

Ineffective Huge Search Space. In the literature, it is commonly assumed that the number of evaluations required to find an optimum is proportional to the size of the *search space* [48]. Popular databases such as PostgreSQL or MySQL have hundreds of configurable knobs [20, 24]. Studies have shown that selecting a few important knobs can expedite the subsequent tuning process [23, 51]. However, a static selection of global important knobs is not applicable to various workloads, since they may vary across workloads [23]. To select workload-specific important knobs, existing practices conduct many target workload runs on different configurations before proceeding to tune the selected knobs. Unfortunately, the number of workload runs required to well select the important knobs could be up to hundreds in quantity, which even surpasses the workload runs required for the actual tuning process itself, as discussed in Section 3.1. As a result, the high cost associated with this approach poses a significant challenge for enhancing the overall tuning efficiency in practice.

We also find that a vital factor to speed up the tuning process has been ignored for long. That is how to define an appropriate value range for each tuning knob. Existing tuning approaches usually adopt the default value ranges provided by the database manual, which are excessively broad and may contain theoretical values that are infeasible for a specific workload. For instance, the default range for `innodb_buffer_pool_size` is 5 MegaByte to 16 ExaByte in MySQL, with the upper bound far exceeding the available instance memory. As a result, tuning efforts would be wasted in unproductive areas [49], leading to out of memory errors [21]. Furthermore, even when the upper bound of memory-related knobs is set below the instance capacity, there is still a significant amount of superfluous space, which also holds true for other types of knobs. This excessive space arises because the default ranges are designed to cover all possible workload scenarios, rather than being customized to a specific workload, so the default ranges are excessively large for performance tuning, as shown in Section 3.2. Given this fact, we propose to build compact ranges that are tailored to each workload. It should be as small as possible while still including optimal regions. In such a way, the search space could be substantially reduced, which further improves tuning efficiency. Unfortunately, identifying the compact knob ranges without lots of target observations is not easy. To this end, we face the challenge that **"how to define a compact space for a given tuning task without the need for a large number of workload runs"**.

Fixed Search Optimizer for Different Tuning Tasks. Many advanced *search optimizers* have been proposed to navigate the search space for database tuning, which can be classified into two categories: search-based and learning-based. Search-based optimizers employ heuristics or meta-heuristics to explore optimal configurations, such as BestConfig [56] and genetic algorithm (GA) [9]. Learning-based optimizers aim to improve exploration by modeling the performance function and can be further classified into Reinforcement-Learning based [9, 31, 50] and Bayesian-Optimization based [5, 10, 15, 17, 24, 27, 53, 55] approaches.

Despite the availability of various *search optimizers*, the applicability of different optimizers remains unclear since no single optimizer can dominate all tuning tasks, as indicated by the no free lunch theorems for optimization [48] and the empirical studies [6, 51] on database tuning. Using an inferior and sub-optimal optimizer could result in a significant performance loss of several orders of magnitude [6, 51]. A recent study [9] on database tuning has suggested that GA is suitable for the early tuning phase, where it focuses more on sampling configurations with high short-term gains, while DDPG performs better in the later stages. Accordingly, it proposes to adopt GA during the early tuning phase and then switch to DDPG [35] for higher performance. However, the real-world scenarios are more complex due to the ever-increasing candidates of *search optimizers* and distinct tuning tasks, e.g., various workloads and different shapes of search space. Simple heuristics fail to recommend the optimal *search optimizer* since they cannot capture the relationship between the tuning tasks and the performance of disparate optimizers. Therefore, to further enhance the efficiency of database tuning, we face the challenge that **"how to identify an appropriate search optimizer for a specific tuning task"**. The decision is hard to make since we cannot perform exhaustive testing of all candidate optimizers on potential tuning tasks, as it is prohibitively expensive.

Our Approach. The aim of this study is to expedite database tuning by simultaneously addressing the aforementioned two challenges by the automation of search space construction and optimizer selection. To achieve this, we propose OpAdviser, a data-driven approach that acts as an Optimization Adviser for database configuration tuning. Specifically, tuning services could accumulate a wealth of historical data as they perform tuning for different clients and applications. Valuable knowledge can be extracted from the historical data to guide the setting of target tuning without conducting extensive experiments. Thus, OpAdviser leverages the data collected from previous tuning tasks and constructed benchmark data to automatically build a compact search space and select an appropriate search optimizer for a given task.

First, OpAdviser constructs a compact *search space* which can significantly reduce the number of target workload runs needed to finish knob tuning. It learns the geometries of search space from the tuning data collected in previous tuning tasks. Although different tasks share common knowledge, their respective important knobs and effective ranges are quite different. Consequently, the geometries derived from one previous task may not be entirely suitable for the target. To address this issue, OpAdviser extracts the promising region from different source tasks based on their similarity with the target task. It adjusts the task similarity during tuning based on the augmented observations and adapts the target search space according to the on-the-fly task similarity through weighted voting. The transfer process is carefully designed so that the negative transfer is avoided when source tasks are less similar and the common geometries are extracted to tailor the search space.

Second, OpAdviser selects a suitable *search optimizer* by capturing the mapping from task characteristics to the performance ranking of each optimizer. It takes an arbitrary task as input and predicts the most promising optimizer without online testing via a

pre-trained model. The difficulty in this process lies in two aspects. The first is how to capture the relationship from various tuning tasks to the performance of different optimizers. The second is how to collect diverse data for training functional models. To learn the relationship, OpAdviser derives meta-features from tuning tasks and employs a sample-efficient pairwise ranking model that can predict the relative performance of two candidate optimizers given an input meta-feature. And to enhance the quality of the training dataset, OpAdviser employs active learning to construct the dataset in the offline phase. To summarize, our contributions include:

- The first work that systematically and efficiently addresses the search space construction and the search optimizer selection problems for database configuration tuning (Section 4).
- A transfer-learning approach that constructs compact search space by extracting the promising geometries from similar historical tasks (Section 5).
- A data-driven method that recommends suitable search optimizers by capturing the relationship from the task characteristics to the performance ranking of candidate optimizers (Section 6).
- A thorough comparison of OpAdviser against state-of-the-art tuners and further analysis that validates our design (Section 7).

The remainder of the paper is organized as follows. We formally define the database configuration tuning problem in Section 2 and analyze the related unsolved problems in Section 3. Then, we provide system overview in Section 4 and introduce space construction in Section 5 and optimizer recommendation in Section 6. We report the evaluation results in Section 7 and conclude the paper.

2 PRELIMINARIES

We formulate the problem and define the related terminologies.

Database Tuning Problem. We consider the scenario where a DBMS is optimized for a target workload. The end-user provides a performance metric to be optimized, denoted by a performance function f. Common performance functions used in database tuning include system throughput, or tail latency (e.g., 95th percentile latency). Given a specific configuration θ, accessing its corresponding performance $f(\theta)$ requires stress testing the DBMS using the target workload when θ is applied. Assuming that the objective is a maximization problem, database configuration tuning aims to find a configuration $\theta^* \in \Theta$ in a given search space Θ, where

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} f(\boldsymbol{\theta}). \tag{1}$$

Search Space. A search space is the set of all possible configurations. DBMS has a set of n configurable knobs $k_1, \ldots, k_n$, alongside their respective domains $\Theta_1, \ldots, \Theta_n$. The knob domains can be either continuous or categorical. A continuous domain is defined by the lower bound l_i and the upper bound u_i. And a categorical domain is defined by the set of possible values s_i. Given the set of knobs, the search space can be defined as $\Theta = \Theta_1 \times \ldots \times \Theta_n$. It can be viewed as an n-dimentional bounding box (or hyperrectangle), which is parameterized by the value range of each knob.

Example. Most database tuning systems prune search space by identifying top-k important knobs. The unimportant knobs are fixed to the default value or the best value observed so far. In this work, we propose to construct a search space $\hat{\Theta}(\Phi)$ by identifying tight value ranges with Φ as a controled parameter. If a knob is continuous, its range parameter $\Phi_i = (\hat{l}_i, \hat{u}_i)$, where $\hat{l}_i \geq l_i$ and $\hat{u}_i \leq u_i$. If a knob is categorical, its range parameter $\Phi_i = \hat{s}_i$, where $\hat{s}_i \subseteq s_i$. We also refer to the reduced ranges as effective ranges.

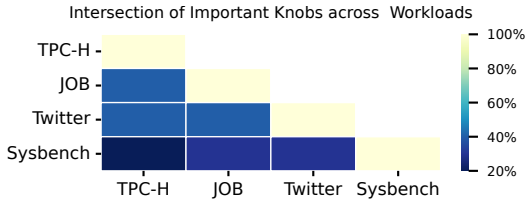


Figure 1: Important Knobs Are Workload-Specific – We present the percentage of intersection among the top-10 important knobs from four different workloads.

Search Optimizer. The search optimizer is the algorithm that suggests a promising configuration over a given search space to improve the pre-defined performance metrics. Due to the increasing interest in configuration tuning, there are many candidate algorithms such as MBO [25], SMAC [22], DDPG [35], GA [29], and so on. We denote the search optimizer adopted at iteration t as O_t.

Example. CDBTune adopts DDPG along the whole tuning process, i.e., $O_t = DDPG$, $t = 1, \ldots, T$. Hunter adopts GA in the first 140 iterations and adopts DDPG afterward, which is denoted by

$$O_t = \begin{cases} GA, & t \leq 140 \\ DDPG, & t > 140 \end{cases} \tag{2}$$

Historical Observations. We denote the historical observations of a tuning task w_i by $H^i = \{\theta^i_j, f(\theta^i_j, w_i)\}_{j=0}^{T_i}$, where $f(\cdot, w_i)$ is the performance function under the tuning task w_i. We denote the target tuning task as w^t with its observations denoted as H^t.

3 PROBLEM ANALYSIS

There are general rules and guidelines to construct the search space and select a search optimizer. But they have limitations: either requiring a large number of target workload runs (contrary to the aim of efficient tuning) or exhibiting unstable performance (inconsistent among different tuning tasks). Hence, we take a deep dive into existing problems and discuss the issues we address.

3.1 Knob Selection

Database systems typically own hundreds of configuration knobs that determine the system's runtime behavior. For instance, Mysql and PostgreSQL have over 190 and 170 configurable knobs, respectively [24]. Research [23, 51] has shown that tuning a small subset of knobs can be sufficient to achieve near-optimal performance and accelerate the configuration optimization due to the reduced search space. In practice, DBAs pre-select several knobs based on their experience and past tuning observations [53]. Regardless of the difference in target workloads, the pre-select workload is tuned. However, important knobs are workload-specific. Figure 1 presents the intersection of top-10 important knobs from four workloads, where the intersection of important knobs between any two workloads is less than 50%. Thus, tuning according to a static ranking would restrictively improve the database performance.

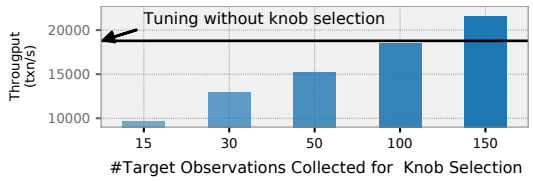


Figure 2: Well Selecting Important Knobs Takes Considerable Observations – Tuning performance on top-10 knobs selected based on different numbers of observations.

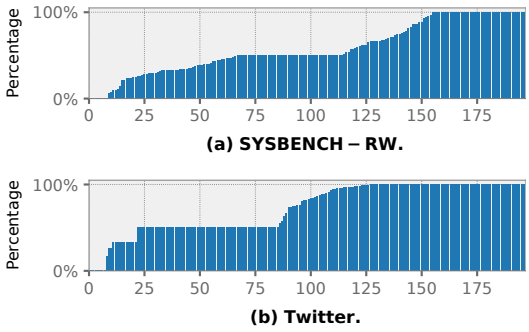


Figure 3: Default Ranges Are Superfluous – The proportion of effective range size against the default range size.

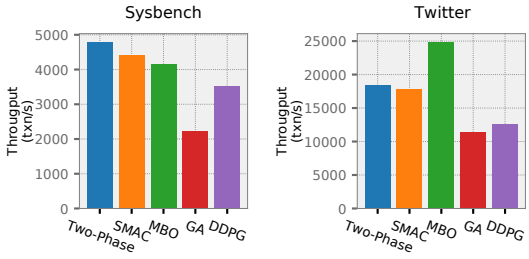


Figure 4: No Single Optimizer Dominates All Tuning Tasks – Performance of different optimizers on two workloads.

To automatically identify workload-specific important knobs, existing approaches [5, 9, 17, 51] analyze knob importance by collecting target observations under different configurations [24]. First, they generate hundreds of configurations using a space-filling sampling technique, such as Latin Hypercube Sampling (LHS) [38], or a search optimizer, such as SMAC [22]. Each configuration is then evaluated in the DBMS by running the target workload, and a machine learning model is trained based on the observations to quantify how each knob affects the DBMS performance. Finally, an optimizer is adopted to optimize over a smaller space defined by the selected knobs with high importance. The preceding knob selection process can accelerate the later optimizations but comes with the cost of collecting hundreds of target observations. Collecting a target observation takes comparable time with a tuning iteration since the time of running target workload dominates both of them. Currently, state-of-the-art methods are required to deliver a satisfying tuning performance within hundreds of iterations (e.g., 200 iterations) [51]. Using the above process to select important knobs for each tuning task clearly breaks the goal of efficient tuning.

To highlight such cost, we generate target observations on Sysbench by sampling configurations using LHS from a 50-dimensional space and evaluating their performance via target workload runs. Following the above knob selection procedure, we select top-10 important knobs based on different numbers of target observations, respectively. Figure 2 presents the performance of tuning the selected knobs where the performance of tuning all the 50 knobs is a baseline. We can see that only when more than 100 observations are used to select the knobs, the tuning performance becomes better than the baseline. Moreover, the knobs selected based on insufficient target observations results in a biased search space that excludes the optimal regions, leading to a bad tuning performance. In summary, selecting knobs based solely on target observations is costly and inefficient. And we aim to identify workload-specific important knobs by leveraging similar historical tasks, rather than relying on a large number of target workload runs.

3.2 Knob Rang Setting

Despite the interest in selecting important knobs, there are currently no established guidelines to set the value ranges of each knob for a given workload. Existing practices adopt the default value ranges provided in the database manuals. These ranges are excessively broad and contain invalid and nonsense values for a specific tuning task, leading to a highly large and necessary search space.

We conduct experiments to analyze the effective ranges that are worth tuning. We collect approximately one thousand observations under various configurations for Sysbench and Twitter workloads and then derive the effective range for each knob based on the observations. The approach for deriving the effective range is presented in Section 5.2. Essentially, only the configurations that fall within the effective range have the potential to outperform the default configurations. The results, presented in Figure 3, demonstrate that more than 100 knobs have narrower effective ranges than the default ranges specified in the MySQL manual. For a fair comparison, we set the default upper bound for memory-related knobs below the instance capacity. Specifically, the space defined by the effective ranges is $4.8 \cdot 10^{79}\times$ and $2.4 \cdot 10^{53}\times$ smaller than the default space for Sysbench and Twitter, respectively. The findings indicate that the conventional practice of exploring the space defined by default ranges is resource-intensive, which wastes a large number of trials exploring infeasible areas.

However, defining appropriate value ranges is a non-trivial task that requires careful consideration. One must balance the need to exclude unpromising regions while avoiding the risk of excluding optimal areas. Additionally, similar to the important knobs, the effective range may differ between workloads. Therefore, pre-derived effective ranges cannot be directly transferred to new tasks without adaptation. As a concrete example, the effective range of `innodb_thread_concurrency` on the Sysbench workload is larger than that on the Twitter workload, whereas the opposite is true for `innodb_spin_wait_delay`. To this end, in this paper, we seek to construct specific value ranges for a given workload.

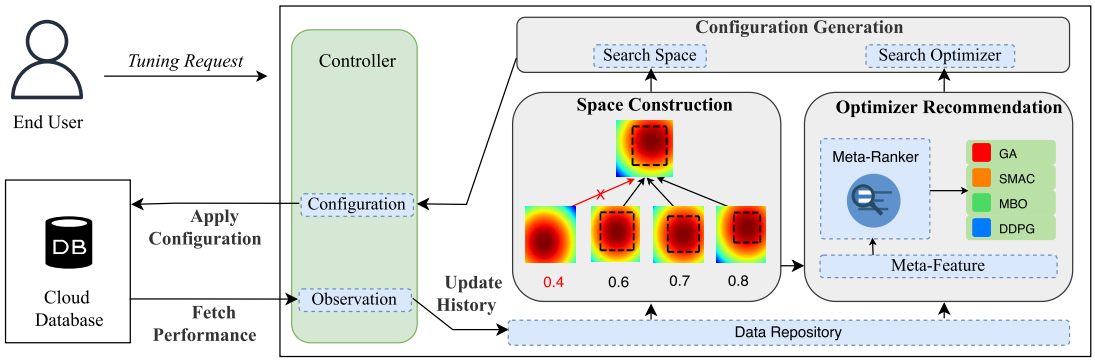


Figure 5: Overview: Architecture and Workflow of OpAdviser.

3.3 Search Optimizer Selection

Given the availability of various search optimizers for database tuning, several rule-based heuristics exist for choosing search optimizers. For instance, Hunter [9] suggests a two-phase approach that initially utilizes GA [29] and subsequently DDPG [35]. And one experimental study [51] has demonstrated empirically that SMAC [22] has the best overall performance among the candidate optimizers. However, faced with various tuning tasks, the general rules could fail and cause bad tuning performance. To illustrate, Figure 4 represents the tuning performance of the two-phase approach and the performance of different optimizers on Sysbench and Twitter workloads. Notably, selecting a suitable strategy can improve the tuning performance (e.g., Two-Phase in Sysbench and MBO [25] in Twitter). Nonetheless, different search optimizers demonstrate varying performance rankings on the two workloads, and the general rules fail to recommend an optimizer that consistently performs the best. Therefore, given the different characteristics of tuning tasks, it is vital but challenging to select suitable search optimizers.

Opportunities. This paper aims to explore two directions: (1) identifying workload-specific important knobs and their effective ranges without requiring extensive target observations, and (2) selecting appropriate search optimizers to enhance tuning efficiency. To achieve these goals, we propose a transfer-learning-based approach that obviates the need for exhaustive experimentation on the target workload by utilizing historical observations from previous tasks. Previous studies in database tuning have utilized historical observations to enhance the performance model of the search optimizer, such as workload mapping [5], RGPE [53], and fine-tuning [50]. Orthogonal to these approaches, we focus on automatically designing search space and selecting search optimizers with the merit of transfer learning.

4 OVERVIEW

Figure 5 presents the architecture and workflow of OpAdviser.

Architecture. The right-hand side represents the client where end users submit tuning requests and a cloud database instance to be optimized. On the right-hand side, the tuning system, OpAdviser, is comprised of five components: (1) controller, which controls the tuning process by interacting with end users and the database instances; (2) data repository, which stores tuning-related data, including historical observations from different tuning tasks. Observations in the data repository are represented as $\{\theta_j^i, f(\theta_j^i, w_i)\}$, where i indicates the i^{th} task. The observations contain the configurations of all configurable knobs, even those that are not tuned during tuning, and therefore have the same dimensions; (3) space constructor, which constructs a compact search space; (4) optimizer adviser, which selects a suitable search optimizer; and (5) configuration generator, where the selected optimizer generates a promising configuration over the constructed search space.

Workflow. To initiate a tuning task, the end user first specifies the tuning objective, tuning budget, the database instance, and the target workload. An iterative tuning workflow is then activated. At each iteration, the **controller** applies a new configuration to the cloud database, executes the workload, and fetches the database performance after the workload running finishes (steps 1-2 in Figure 5). The observation $\{\theta_j^t, f(\theta_j^t, w_t)\}$ is then stored in the **data repository** (step 3 in Figure 5). Using the observations of the target and historical tasks, **space constructor** (discussed in Section 5) identifies similar tasks and constructs a compact search space by combining their promising regions (step 4 in Figure 5). Then, **optimizer adviser** (discussed in Section 6) extracts a meta-feature from the target task by analyzing its search space and historical observations, and inputs the meta-feature to a meta-ranker, which is pre-trained on constructed benchmark data. Then the meta-ranker predicts the performance ranking of the candidate optimizers, and a top-ranked optimizer is recommended (step 5 in Figure 5). Subsequently, the recommended optimizer and the constructed search space are passed to **configuration generator** where the optimizer explores the search space by suggesting a promising configuration, which is passed to the **controller** (step 6 in Figure 5). Once the tuning budget is exhausted, OpAdviser returns the best configuration found so far to the end user. OpAdviser is designed to be self-improving, whereby its performance improves with the increasing tuning observations accumulated in the **data repository**.

5 SEARCH SPACE CONSTRUCTION

In this section, we present a new approach to automatically construct a precise search space without collecting large numbers of observations on the target workload. Our approach takes the observations from previous tasks (also referred to as source tasks)

and the target task as inputs and outputs a compact target search space $\hat{\Theta} \subseteq \Theta$. Since $\hat{\Theta}$ is smaller, the search optimizer will find its optimum faster, i.e., requiring fewer workload runs. To construct $\hat{\Theta}$ without many target observations, we utilize the effective regions extracted from similar source tasks. Our method has three steps: (1) similar task identification where we identify a set of candidate tasks with similar characteristics to the target task, (2) effective region extraction where we extract the effective region from each candidate based on the task similarity, (3) majority weighted voting which utilizes the geometries of the extracted regions to generate the target space where the majority regions are retained.

5.1 Similar Task Identification

A database tuning system can accumulate a wealth of historical observations as it performs different tuning tasks. These previous tasks can offer valuable insights for identifying the promising region for target tuning. For instance, similar workloads running on different hardware may share common ranges for hardware-agnostic knobs. Even for different workloads, the bad search area could be similar, such as a buffer size that is too small. Hence, for a given target task, examining similar historical tasks could reveal analogous promising regions.

The first question is how to quantify the similarity between a source task and the target task so that we can transfer the effective regions accordingly. We address this problem by considering whether the two tasks have similar performance rankings for different configurations, which indicates whether we can adopt the good region of one task for another. For the target task and a source task, we measure their similarity as the ratio of *concordant ranking pairs* based on the target observations. In this context, given two target observations $(\theta_1, f(\theta_1))$ and $(\theta_2, f(\theta_2))$, if a the performance model of the source task can correctly predict their relative performance, we identify the presence of a "concordant ranking pair" between the source and target tasks. The ratio of *concordant ranking pairs* reflects the generalization ability of source performance model on the target task. Formally, we define the similarity $S(i,t)$ between source task w_i and the target task w_t as follows:

$$\begin{aligned} S(i,t) &= \frac{2}{|H^t|(|H^t|-1)} F(i,t), \\ F(i,t) &= \sum_{j=1}^{|H^t|} \sum_{k=j+1}^{|H^t|} \mathbb{1}\Big(f(\theta_j) \le f(\theta_k)\Big) \otimes \mathbb{1}\Big(f^{'}(\theta_j, w_i) \le f^{'}(\theta_k, w_i)\Big), \end{aligned} \tag{3}$$

where $F(i,t)$ is the number of *concordant ranking pairs*, H^t is the number of target observations, and $\otimes$ is the exclusive-nor operator. And $f^{'}$ is performance model fitted on the historical observations of the source task w_i. We train the performance model in the offline phase using the random forest algorithm as it is lightweight and known to perform well for categorical input data [22].

Taking Figure 6 as an illustration, the first row represents the target response surface, while the subsequent three rows depict the source response surfaces. In the 4th iteration, we have three target observations ($\theta_1 < \theta_2 < \theta_3$) with performance ranking pairs: $f(\theta_1) \le f(\theta_2)$, $f(\theta_2) \ge f(\theta_3)$, and $f(\theta_1) \le f(\theta_3)$. Considering the source task w_3, its rankings for these three configurations are: $f^{'}(\theta_1) \le f^{'}(\theta_2)$, $f^{'}(\theta_2) \ge f^{'}(\theta_3)$, and $f^{'}(\theta_1) \ge f^{'}(\theta_3)$. It exhibits two concordant pairs with the target, resulting in $S(1,t) = \frac{2}{3}$. Similarly, in the 6th iteration, there are 10 performance ranking pairs. Task w_3 can correctly rank three of them, resulting in $S(3,t) = \frac{3}{10}$. As the number of target observations grows, the ratio of concordant ranking pairs increasingly approximates the ground-truth similarity. OpAdviser filters source tasks with similarity below 0.5 because they provide less utility for the target than a random guess.

5.2 Effective Region Extraction

In this subsection, we explain how we extract the effective region from a source task by taking into account its similarity to the target task. We denote the region extracted from i-th task as $\hat{\Theta}(\Phi^i)$, where Φ is the parameter that defines the effective region, as formulated in Section 2. Our goal is to obtain an effective region that contain the promising configurations while keeping it as small as possible for the sake of efficient tuning. To this end, we formulate the identification of the effective region as a constrained optimization problem, where we minimize the size of the effective region while ensuring that it still includes promising configurations:

$$\begin{aligned} &\arg\min_{\Phi^i} \Lambda(\Phi^i), \\ &\text{s.t.} \quad \forall \theta \in G, \theta \in \hat{\Theta}(\Phi^i). \end{aligned} \tag{4}$$

In Equation 4, the objective function minimizes the size of $\hat{\Theta}(\Phi^i)$, while the constraint condition ensures that $\hat{\Theta}(\Phi^i)$ contains the promising configurations. Specifically, $\Lambda(\Phi^i)$ denotes the size of $\hat{\Theta}(\Phi^i)$, and G denotes a set of promising configurations. We consider the configurations with performance level above a certain threshold as promising, i.e., $G = \Big\{\theta \in \Theta | f(\theta, w^i) \ge f_b^i\Big\}$, where f_b^i is the performance standard. To form G, we utilize the observed configurations in H^i and randomly sampled configurations. For an observed configuration, if its observed performance $f(\cdot, w_i)$ is higher than f_b^i, we add it to G. For a randomly sampled configuration, if its predicted performance $f^{'}(\cdot, w_i)$ is higher than f_b^i, we add it to G. And the size of $\hat{\Theta}(\Phi^i)$ is quantified by the total number of possible configurations within the region:

$$\Lambda(\Phi^i) = \prod_{k_j^i \in K1} \Big(\hat{u}_j^i - \hat{l}_j\Big) \prod_{k_t^i \in K2} \Big|\hat{s}_t^i\Big|, \tag{5}$$

where $K1$ denotes the set of continuous knobs and $K2$ denotes the set of categorical knobs. To this end, we can derive that Equation 4 has a simple closed-form solution. For a continuous knob k_j, its lower bound and upper bounds are given by

$$\hat{l}_j^i = min\Big\{\theta_j \Big| \theta \in G\Big\}, \quad \hat{u}_j^i = max\Big\{\theta_j \Big| \theta \in G\Big\}. \tag{6}$$

For a categorical knob k_t, its possible value is given by

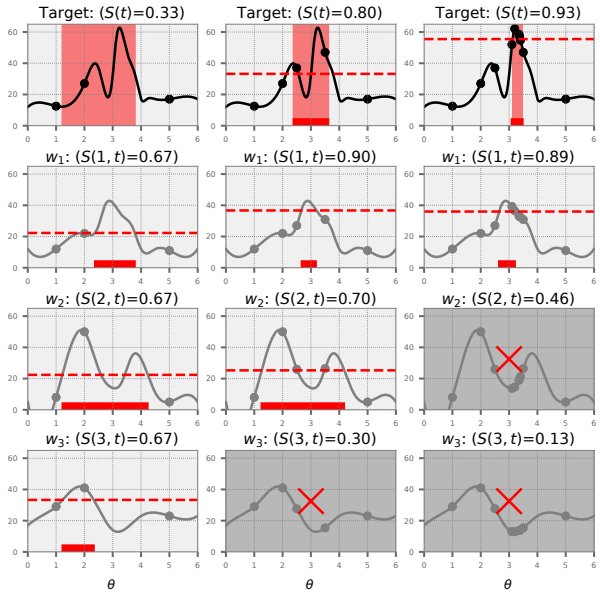
$$\hat{s}_t^i = \Big\{\theta_t \Big| \theta \in G\Big\}. \tag{7}$$

The effective region encompasses all the potential configurations with performance superior to f_b^i. Given the solution of Equation 4, it is not difficult to see that the value of f_b^i largely affect the size of the effective region. To properly control the region size, we adjust f_b based on task similarity, as defined by Equation 8::

$$f_b^i = f_{default}^i + 2\Big(f_{best}^i - f_{default}^i\Big)\Big(S(i,t) - 0.5\Big), \tag{8}$$

where $f^i_{default}$ is the default performance of the source task, f^i_{best} is the best-observed performance of the source task, and $S(i,t)$ is the task similarity between the source and the target task. The underlying principle of this design is as follows. If the source task is highly similar to the target task, f^i_b approaches f^i_{best}, resulting in a small extracted region. Otherwise, f^i_b deviates from f^i_{best}, leading to a large extracted region with less space pruned. Continue with the running example in Figure 6, with $\theta = 1$ as the default configuration. In the 6th iteration, we have $f^1_b = f^1_{default} + 0.6(f^1_{best} - f^1_{default})$ and $f^2_b = f^2_{default}$. Consequently, this leads to a smaller effective region (red area) when utilizing w_1 as the basis and a larger one when using w_2. Given the high similarity between w_1 and the target, we can extract a small region based on w_1 with a high confidence that the optimal configuration of the target lies within it. Conversely, due to the lower similarity of w_2 to the target, OpAdviser controls the extracted region based on w_2 to be relatively large. This prevents aggressive pruning of the search space, such as in the case of pruning the region $3.5 < \theta < 4.5$ based on w_2.

To further decrease the dimensionality of the effective region, we employ feature selection algorithms [19] to identify important knobs based on the historical observations H^i. Specifically, we adopt SHAP [36], a toolkit that measures feature importance by interpreting performance changes between configurations. It has been shown to yield the most meaningful importance scores in the context of database tuning [51]. For each knob, SHAP computes its contribution (i.e., SHAP value) to the enhancement of the system's performance from the base level to the target level. We then discard the knob with only negative contributions (assuming a maximization objective). The removal of a knob indicates that it does not merit further tuning, and its effective range is set to zero.



(a) Iteration 4. (b) Iteration 6. (c) Iteration 11.

Figure 6: A Toy Example, Showing Three Steps of Space Construction: (1) Computing similarity $S(i,t)$ based on the number of concordant ranking pairs between the target performance (black point) and the predicted performance (grey point) and filtering tasks with $S(i,t) < 0.5$ (red cross). (2) Extracting effective regions (red area) with performance better than a standard (red dashed line) controlled by $S(i,t)$. (3) Constructing target range (pale pink shading in the first row) by a weighted combination of the red areas.

5.3 Majority Weighted Voting

Given the effective regions extracted from each candidate task, the final step of our approach is to generate the target search space, including the important knobs and their value ranges.

We adopt a majority weighted voting strategy to aggregate the suggestions from candidate tasks, where the the decision of majority is respected. The weight assigned to i-th task is proportional to its similarity with the target task, i.e., $w_i = \frac{S(i,t)}{\sum_{j=1}^{m} S(j,t)}$, where m is the number of voters. To ensure consensus among the candidate tasks, we set the total weight threshold to 50% [41] . Knobs with a majority agreement to remove are eliminated, while those with a majority agreement to retain are preserved. Importantly, unlike existing practices [9, 51, 53], this approach does not require a hyperparameter for setting the number of important knobs. Then, for each retained knob, we enumerate the extracted effective ranges and also retain the parts with the majority vote. We generate the target range by creating one tight bounding box enclosing all the regions with majority voting.

To further avoid negative transfer [57], we employ two strategies. First, we consider the target task as a voter. We generate a referenced effective region using the current target observations, using the same approach in Section 5.2. To calculate the weight of the target voter, we measure its generalization ability on unseen configurations using the out-of-sample concordant ordering ratio, defined in Equation 9:

$$
\begin{aligned}
S(t,t) &= \frac{2}{|H^t|(|H^t|-1)} F(t,t), \\
F(t,t) &= \sum_{j=1}^{|D^t|} \sum_{k=j+1}^{|D^t|} \mathbb{1}\Big(f(\theta_j) \le f(\theta_k)\Big) \otimes \mathbb{1}\Big(f'_{-j}(\theta_j) \le f'_{-k}(\theta_k)\Big),
\end{aligned}
\tag{9}
$$

where $f'_{-j}(\theta_j)$ is the performance prediction of θ_j from the model fitted on H^t with the observation for θ_j left out. As the target observations increase with each iteration, the target model's generalization ability improves steadily. OpAdviser extracts the effective region based on the target task, when the leave-one-out model can better generalize to unseen configurations than random guess, i.e., $S(t) > 0.5$. In the ongoing example illustrated in Figure 6, OpAdviser warms up the space recommendations using the historical tasks in the 4th iteration. The target model is introduced since the 6th iteration with its weight increasing, so that the geometries learned from target observations are given more importance. Second, we sample voters from the candidate tasks instead of using all of them. We denote the set of candidate tasks as C. We sample k ($k < |C|$) tasks out of the candidate tasks without replacement, following the possibilities based on their scaled similarities. The task sampling adds randomness to the generated target space and avoids the optimizer being trapped in the local optimum.

6 SEARCH OPTIMIZER RECOMMENDATION

In this section, we present a data-driven approach that recommends suitable search optimizers for a given task. Instead of rule-based selection, our approach considers the task characteristic and suggests a promising optimizer accordingly. It leverages the knowledge extracted from the running history of different optimizers through a pre-trained model and can thus produce predictions without actually running candidate optimizers on the current tasks.

6.1 Meta-Feature Extraction

In the context of optimizer selection, the meta-feature is informative tuning-related variables that characterize the tuning task and can impact the tuning performance of different search optimizers. Studies have shown that several factors in a tuning task can lead to different suitable optimizers. For example, the dimensionality and composition of a search space can impact the performance of search optimizers, with some optimizers being better suited to more categorical or continuous knobs [51]. The response surface of the database performance function can also impact the search performance of optimizers [51]. The response surface is the mathematical relationship between the input variables and the output function. In the context of database tuning, the difference in response surface is mainly caused by different workloads and hardware environments. We also consider the tuning phase, as some search optimizers have been shown to perform better in the early stage of tuning, while others perform better later [9]. In summary, We extract the following meta-features for each tuning task:

(1) Space feature. This feature includes the number of tuning knobs, the size of the search space, and the ratio of continuous tuning knobs to categorical tuning knobs.
(2) Response surface feature. We adopt a similarity vector that contains the ratios of concordant ordering pairs (discussed in Section 5.2) between the current task and each previous tuning task. This vector can be viewed as a distributed representation of the target response surface [12].
(3) Tuning process feature. We use the number of current iterations as the feature to represent the tuning process.

The meta-feature is updated after each tuning iteration so that we can select suitable optimizes for different iterations. Note that different search optimizers could share the same tuning observations, providing us with the opportunity to adopt different optimizers during the tuning process.

6.2 Offline Data Generation

It is challenging and unreliable to rely on general heuristics, such as "DDPG performs well in large search spaces [50]", "SMAC is the overall best [51]", and "start with GA before switching to DDPG [9]" for optimizer selection. One reason is that determining suitable decision thresholds, such as at which size DDPG is preferred over SMAC or the iteration count for transitioning from GA to DDPG, is complex. And choosing the appropriate heuristic to follow can be difficult. Additionally, the heuristics rely on human experience, which may not cover the diverse tuning tasks with varying features.

To address these challenges, we propose a data-driven solution utilizing machine learning models for tuning task selection. In order to train our model, we require data that demonstrates how different candidate optimizers perform under specific tuning conditions. However, performing exhaustive testing on all the potential tuning tasks is prohibitively expensive, given so many possible tuning conditions. To efficiently collect the data with less testing effort, we employ active learning techniques to select samples for testing and labeling. We begin by generating a set of candidate tuning tasks by varying search spaces and response surfaces. We iteratively choose tasks with the highest uncertainty to test, as indicated by the classification margin [30], a well-established metric for active learning. The sample with the smallest margin represents the greatest uncertainty. This process continues until a desired decision margin level is reached or the testing budget is exhausted.

6.3 Meta-Ranker Construction

Utilizing the collected data, we proceed to build our learning model, referred to as namely meta-ranker. Our approach frames optimizer selection as a ranking problem, with the objective of predicting the pairwise ranking of optimizers based on provided tuning conditions. To achieve this, we employ LambdaMART [8], a well-established learning-to-rank algorithm. LambdaMART utilizes gradient-boosting decision trees to optimize a pairwise loss function, effectively capturing the relative performance of optimizers in diverse tuning scenarios. Meta-ranker takes as input the task meta-feature and two candidate optimizers for comparison, producing a ranking of their relative performance on the given task. Its training data is structured as (m, o_i, o_j, I), where m represents the meta-feature of a tuning task, o_i and o_j are one-hot encodings for candidate optimizers, and I serves as an identifier. When o_i outperforms o_j, I is assigned the value of one, and zero otherwise.

In the online phase , OpAdviser extracts the meta-feature from the target task, inputs this feature along with the pair of candidate optimizers into meta-ranker, and recommends the top-ranked optimizer. The meta-ranker allows us to systematically discern optimizer performance concerning specific tuning task characteristics. And it offers the advantage of being more sample-efficient than other models [18, 34]. Compared to a direct classification model to predict the best optimizer, more information can be extracted by considering pairwise performance. Moreover, compared with a regression model, the meta-ranker only requires comparisons between pairs of optimizers and handles the noisy data, and data with different scales better.

7 EXPERIMENTS

We first compare OpAdviser's performance against state-of-the-art baselines. Then, we conduct micro-analysis of OpAdviser's individual components and evaluate the generalization ability. Finally, we evaluate on a cold start scenario.

7.1 Experiment Setting

Hardware. Our experiments are conducted on cloud instances, running MYSQL 5.7 as the target DBMS. Unless otherwise stated, we adopt the instance type with 16 cores of CPU and 32 GB memory.
Target Workload. We employ four workloads with different characteristics: Sysbench read-only (RO), Sysbench write-only (WO), Sysbench read-write (RW), and Twitter. We load 300 tables under

Sysbench, with each table containing 800 thousand records, resulting in a total data size of ~ 50 GB. We load the data with a scale factor of 3000, resulting in a total data size of ~ 16 GB. We use throughput as a maximization objective for the four workloads. The above setting is similar to prior works [24, 51].

Candidate Optimizer. Following our prior work [51], we select four promising optimizers, namely MBO [25], SMAC [22], DDPG [35], and GA [29], as candidate optimizers in the optimizer adviser module. MBO is a BO-based optimizer that adopts a mixed-kernel GP as its surrogate model. SMAC adopts a random-forest-based surrogate which assumes a Gaussian model. DDPG adopts Deep Deterministic Policy Gradient algorithm to learn the configuration tuning policy for DBMS. GA is a meta-heuristic inspired by the process of natural selection.

Data Repository. We generate training data for meta-ranker by constructing various tuning conditions. To create the potential tasks, we construct the search space by adjusting the tuning knobs, resulting 390 distinct spaces. Additionally, we diversify the response surfaces using 9 workloads, including Twitter, YCSB, TPC-C, Voter in OLTBench [14], Sysbench RO, Sysbench WO, Sysbench RW, JOB [28], TPC-H [1]. The outlined procedure generates 3510 tuning tasks under varying conditions. Among these, we actively labeled 48 tasks, documenting the tuning performance of the four candidate optimizers across different tuning iterations. Additionally, these tuning observations also used for constructing the search space during the target tuning. To ensure the fairness of comparison, we hold out any observations of the target workload that are present in the data repository. Specifically, when tuning Sysbench, regardless of its mode, we hold out the observations for Sysbench RW, Sysbench RO, and Sysbench WO.

Tuning Setting. MYSQL 5.7 has 197 configurable knobs [51]. OpAdviser automatically constructs a search space from the input 197-dimensional space by identifying important knobs and their effective ranges. For the baselines, we adopt the initial input search space as stated in their papers: the input space contains 90, 65, 20, 197 generally important knobs for LlamaTune, Hunter, ResTune, and CDBTune, respectively. OtterTune incrementally selects important knobs, beginning with 4 knobs. To select the generally important knobs, we rank the knobs by their average SHAP value across the workloads. We mostly adopt the knob ranges provided in the MySQL manual [2, 3], except that if a knob's upper bound exceeds the instance capacity, we set it equal to the instance capacity. We perform three tuning sessions for each baseline, reporting the median of the best tuning performances so far. Each tuning session comprises 200 iterations, where each iteration involves a three-minute stress test to run the target workload.

7.2 End-to-end Comparison

We compare OpAdviser with five existing tuning approaches, namely, LlamaTune [24], Hunter [9], DB-BERT [45], ResTune [53], and SMAC [22]. We also evaluate two ablation versions of OpAdviser: OpAdviser-w/o-Optimizer, which removes the optimizer adviser, and OpAdviser-w/o-Space, which removes the space constructor. Figure 7 illustrates the performance of the best configurations found so far by different approaches throughout the iteration process.

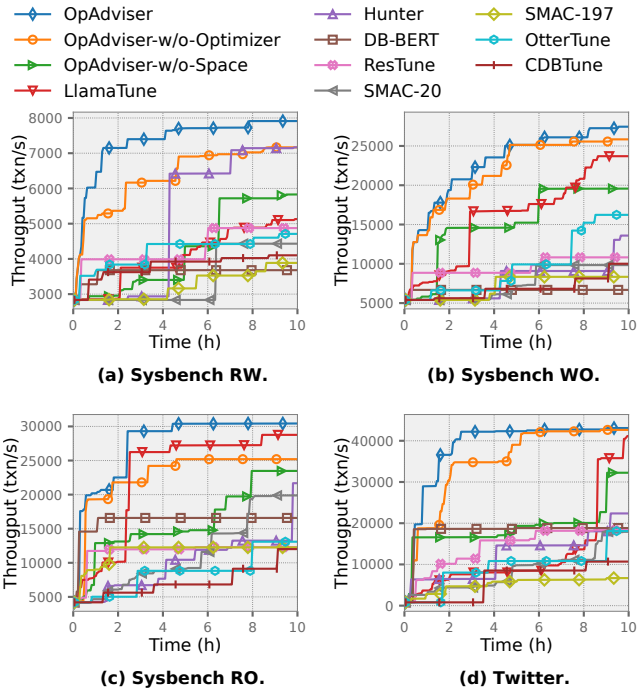


Figure 7: End-to-end Comparison over The Tuning Time.

7.2.1 Comparison with LlamaTune [24]. LlamaTune is a recently proposed database configuration tuning system that reduces the space dimension based on randomized embeddings [39]. It maps the input search space Θ_D to a lower-dimensional intrinsic space Θ_d, where $d << D$, by a randomized projection matrix $A \in \mathbb{R}^{D*d}$, and then optimizes over the low-dimensional Θ_d using the SMAC optimizer. The suggested point p in the low-dimensional space is projected to $\hat{p}$ in the original space by $\hat{p} = Ap$.

OpAdviser outperforms LlamaTune on the four workloads, achieving throughput improvements of 11.24% ~ 452.21% and 7.38% ~ 54.23% given a time budget of 2.5 hours and 10 hours, respectively. We observe that LlamaTune has inferior tuning performance in the initial phase since it does not use prior tuning knowledge (cold start). Additionally, its space projection restricts a one-to-many mapping from Θ_d to Θ_D, forcing multiple knob values to be determined by one dimension in Θ_d. This approach relies on the assumption that tuning a small set of important knobs, whose size is less than or equal to d, can improve database performance. However, values of other less important knobs still need to be located in reasonable ranges, which may be violated by the one-to-many mapping. Therefore, the space projection may cause a ceiling effect on the tuning performance, as shown in Figure 7 (a).

7.2.2 Comparison with Hunter [9]. Hunter is another state-of-the-art database configuration tuning system that divides the tuning process into two phases. In the first 140 iterations (the first phase), it uses GA as the search optimizer to optimize over the input search space and then uses a random forest model to identify the top-20 important knobs based on the observations from the first phase. In the second phase, it uses DDPG as the search optimizer to tune

the top-20 knobs. However, similar to LlamaTune, Hunter also has a cold start issue since it does not utilize prior tuning data. Additionally, its choice of optimizer and the number of iterations using GA are decided empirically based on experiments on several workloads [9], which may not be generally applicable.

7.2.3 Comparison with DB-BERT [45]. DB-BERT applies BERT model [13] to analyze database manuals and other relevant text documents. It fine-tunes model weights in the training phase to translate natural language hints into recommended settings. At run time, DB-BERT uses reinforcement learning to learn to aggregate, adapt, and prioritize hints for optimal performance in a specific tuning task. As shown, DB-BERT recommends better configurations at the early phase, but its effectiveness diminishes later, restricted by the limited tuning hints extracted from texts.

7.2.4 Comparison with ResTune [53]. ResTune fits multiple base learners on observations of each corresponding source task and combines their predictions as a meta-learner. However, the performance scales of different history tasks usually differ, and the combined prediction will be dominated by the base learners where the output values are larger [7]. While ResTune performs well initially due to the use of prior knowledge, it improves slowly later. OpAdviser addresses the scale problem by transferring the effective regions instead of the model's predictions.

7.2.5 Comparison with OtterTune [6]. OtterTune identifies the most similar workload from the data repository through workload mapping and utilizes matched data and target observations to train a surrogate model. OtterTune's approach of merging data without considering their similarity ratio can cause negative transfer. Furthermore, OtterTune selects important tuning knobs using an increasing heuristics approach, which performs less effectively than OpAdviser, as demonstrated in Section 7.4.1.

7.2.6 Comparison with CDBTune [50]. CDBTune employs the data repository's observations to pre-train a DDPG model. Then, it updates the model through gradient descent (fine-tune) based on the feedback information. CDBTune's performance is not stable due to the potential overfitting of the neural network to source workloads.

7.2.7 Comparison with SMAC [22]. We apply SMAC to optimize a search space with 20 generally important knobs (SMAC-20) and all 197 knobs (SMAC-197). SMAC-197 performs poorly because of its excessively large search space. Although SMAC-20 achieves better performance, it still falls behind.

7.2.8 Summary. Compared to the best baseline in each workload, OpAdviser reaches its best configuration ~3.4× faster on average and can achieve the same best throughput with half the tuning budget. It also improves final throughput by ~9.2% on average.

7.3 Overhead Analysis.

Offline Training. OpAdviser collects the training data for meta-ranker through an offline process. In our experimental setup, which involves a 200-iteration tuning budget with 3-minute stress-testing per iteration, the labeling process for one task consumes ~ 2400 minutes (4×200×3). To label all 48 tuning tasks, we allocate approximately 20 days of effort, utilizing four parallel database instances.

Table 1: Algorithm Time Per Iteration on Average.

OpAdviser	LlamaTune	Hunter	DB-BERT	ResTune	OtterTune	CDBTune
5.92s	1.67s	0.02s	9.61s	6.86s	6.89s	0.13s

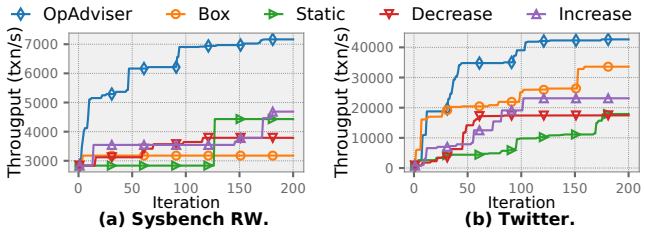


Figure 8: Comparison of Different Space Strategies.

For cloud service providers or companies offering database tuning services, this overhead is manageable and can be quickly recouped by the OpAdviser's tuning efficiency compared to other systems, as evidenced in Figure 7. And the corresponding observations can aid in the construction of the search space. Nonetheless, the offline data collection may pose challenges for small end-users seeking to optimize their own applications. They usually begin tuning with no initial historical data available and we further discuss this cold start scenario in Section 7.7.

Online Generation. We measure the time required by different approaches for generating a configuration, which does not include the time for target workload runs. The results are presented in Table 1. Due to the need of constructing space and selecting optimizers, OpAdviser has a longer algorithm time than several approaches. Specifically, OpAdviser takes 4.93 seconds to construct the search space, with an average time of 4.21 seconds spent on calculating the concordant ranking pairs ratio between the current task and historical tasks. Optimizer recommendation tasks 0.25 seconds where extracting the meta-feature and the inference of meta-ranker are relatively faster. Overall, OpAdviser's algorithm time is comparable to other approaches. Database tuning systems replay the target workload on the suggested configuration to gather the feedback, which usually takes 3 minutes or more to obtain stable observations [6, 50, 53], dominating the tuning time. Therefore, despite OpAdviser's longer algorithm time, OpAdviser still outperforms the baselines in the end-to-end tuning time by identifying better configurations with less target workload runs.

7.4 Evaluation of Space Construction

7.4.1 Comparison with Different Space Construction Strategies. We compare OpAdviser's space constructor with the following baselines: (1) Static, which constructs a search space defined by the top-20 knob based on a static ranking of general importance; (2) Increase, adopted in OtterTune [5], which begins with tuning the top four knobs and adds two knobs every four iterations, according to the static ranking; (3) Decrease, adopted in Tuneful [17], which begins with tuning all the knobs and removes 40% unimportant knobs every 10 iterations based on the importance ranking generated from the previous 10 observations; (4) Box [40], which suggests

Table 2: Ablation in Space Construction on Sysbench RO.

Module			Throughput	Improvement
Knob	Range	Similarity	(txn/sec)	ratio
-	-	-	12240	-
✓	✓	-	9742	-20.4%
✓	-	✓	15872	29.7%
-	✓	✓	21376	74.6%
✓	✓	✓	25406	107.6%

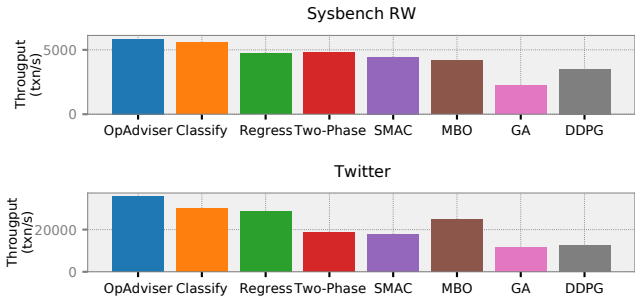


Figure 9: Comparison of Optimizer Selection Strategies.

a minimal search space containing the best-observed configurations of all the previous tasks. OpAdviser outperforms all the baselines, as shown in Figure 8. Static and Increase suffer from the common drawback of excluding workload-specific important knobs from the search space. Although Decrease aims to generate an importance ranking specific to the target task, the 10 target observations are insufficient to accurately rank the knobs. Moreover, these three baselines adopt the default range which is too large, leading to an inefficient search. Box is most closely related to our approach since they both focus on defining more compact value ranges. Nevertheless, Box neglects task differences and merely transfers the space containing best source configurations, resulting in overly aggressive pruning of the search space. By considering task similarity and dynamically adjusting the size of the extracted regions accordingly, our approach effectively addresses the limitations of Box.

7.4.2 Ablation Study. We present an ablation study on the techniques employed in the space constructor: knob selection (Knob), range reduction (Range), and similarity control (Similarity). Tables 2 show the results, where the improvement ratio against using no technique (the first row) is presented. We observe that applying any combination of Similarity with Knob or Range improves performance compared to using no technique. The best performance is observed when all three techniques are used in tandem, validating the efficacy of similarity control. We further observe that the absence of similarity filtering and voting results in negative transfer with an improvement ratio, due to its need for a workload-specific space construction. Besides, we find that range reduction leads to more performance improvement than knob selection.

7.5 Evaluation on Optimizer Recommendation

7.5.1 Comparison with Different Recommending Strategies. We present a comparative analysis of several strategies for recommending optimizers, including: (1) employing a single optimizer, such as SMAC, MBO, DDPG, and GA; (2) OpAdviser, our approach that adopts a pairwise ranking model to predict the relative performance of two candidate optimizers; (3) Classify, a variant of OpAdviser that adopts a classification model to directly predict the best-performing optimizer; (4) Regress, another variation of OpAdviser that employs a regression model to predict the absolute performance of candidate optimizers and recommends the one with the highest prediction; (5) Two-Phase, an approach proposed by Hunter [9], which employs GA in the first 140 iterations and then switches to DDPG. To evaluate the performance of these strategies, we conduct experiments on Sysbench RW and Twitter, and the results are presented in Figure 9. We observe that OpAdviser outperforms the other strategies. Furthermore, in terms of the choice of meta-models, the pairwise ranking model stands out due to its sample efficiency.

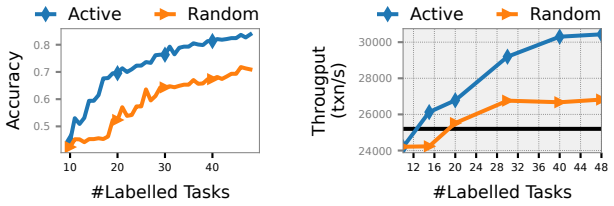


Figure 10: Utility of Active Learning. In Figure (a), we present the accuracy of the meta-ranker trained via active learning and random sampling. Figure (b) displays the associated tuning performance, with a horizontal line representing the baseline performance with SMAC as the default optimizer.

7.5.2 Utility of Active Learning. OpAdviser employs an active learning technique to collect training data for the meta-ranker. In Figure 10, we compare this technique with random sampling, illustrating their classification accuracy for the best optimizer in leave-one-out validation and the tuning performance in SYSBENCH RO. The meta-ranker, trained with active learning, achieves higher accuracy due to learning from more informative data within the same labeling budget. Figure 10 (b) shows that the meta-ranker makes better decisions than the heuristics of "SMAC is the best-performing optimizer in most cases" (indicated by the black horizontal line) after training on the initial 15 labeled tasks using active learning. Moreover, given the same labeling budget, active learning leads to a higher tuning performance than random sampling.

7.6 Evaluation on Generalization

While prior experiments analyze the adaptability to unseen workloads by holding out the target workload, we next evaluate the generalization on different data sizes and hardware settings.

7.6.1 Varying Data Sizes. The appropriate range of a knob could be influenced by the size of the database. We verify the performance of OpAdviser when the target database size diverges from that of the historical tuning tasks. We conduct comparative analyses with LlamaTune, Hunter, and OtterTune using the SYSBENCH benchmark. The data repository includes tuning tasks with database sizes ranging from 10 GB to 40 GB. We load the target database with different sizes by adjusting the number of tables and table sizes. Figure 11 presents results, depicting performance improvements within

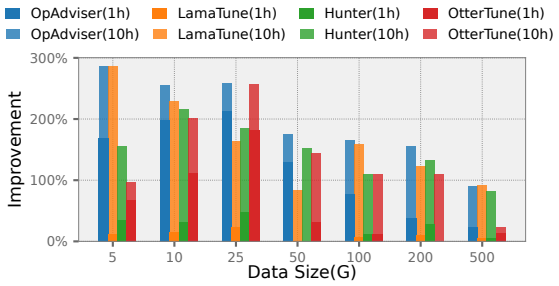


Figure 11: Generalization across Different Data Sizes.

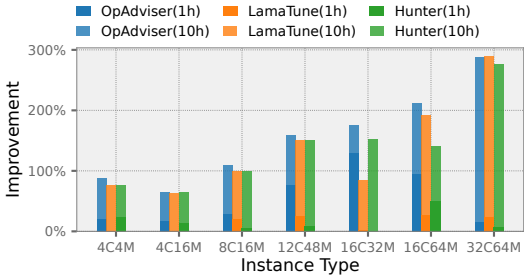


Figure 12: Generalization across Hardware Environments.

1 hour (hatched bars) and 10 hours (solid bars). OpAdviser consistently excels across data sizes, particularly under a 1-hour tuning budget, since it could exclude unpromising regions in the early phase, informed by geometries learned from similar workloads.

To reason the generality across database sizes, we analyze the search space suggested by OpAdviser. Across target data sizes from 10 GB to 50 GB, the suggested space consistently covers the global optimum while pruning unpromising regions throughout tuning iterations. OpAdviser assigns higher concordant ranking ratio to tasks with similar sizes, resulting in compact effective regions. In cases where the target task significantly differs from source tasks, while OpAdviser initially excludes the optimum, it still eliminates unfavorable ranges for lots of important knobs, such as *innodb_thread_concurrency* for concurrency control and `binlog_case_size` for background processes. Notably, OpAdviser progressively expands the search space to encompass optimum, avoiding negative transfer. For instance, in target tasks with a 500 GB database, the optimal value for `table_open_cache` is 3.7k, with a default range of 1 ~ 524k. Initially, at the 10th iteration, OpAdviser suggests a range of 0.9k ~ 2.4k. After 20 iterations, the suggested range expands to 0.5k ~ 5.1k, encompassing the value of 3.7k. This expansion occurs as the concordant ranking ratios approach ground-truth similarity, and the weight of target tasks increases with more target observations accumulated.

7.6.2 Varying Hardware Environments. We assess OpAdviser's generalization across diverse hardware instances, comparing it to LlamaTune and Hunter. OtterTune is excluded from the comparison due to its lack of hardware adaptation support [5]. Figure 12 displays the results, where "4C16M" represents an instance equipped with 4 core CPUs and 16 GB memory, and similar notations apply

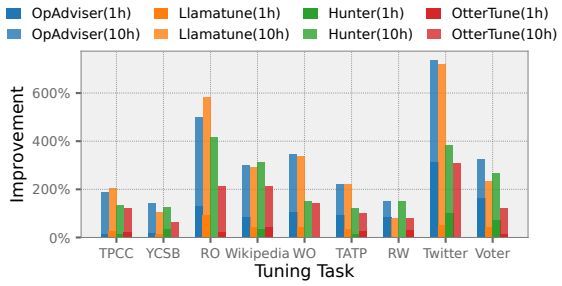


Figure 13: Evaluation on Cold Start Scenario: OpAdviser's advantage increases as completing more tuning tasks.

to other instances. OpAdviser's data repository only contains historical observations for instances with 16 cores CPUs and 32 GB memory. Despite this limitation, OpAdviser consistently outperforms the baselines across configurations ranging from "4C16M" to "16C64M". For instances with more significant differences, OpAdviser still achieves competitive performance with the baselines.

7.7 Evaluation on Cold Start Scenario

We assess OpAdviser's performance under an initial absence of historical data and conduct comparisons with OtterTune, LlamaTune, and Hunter, tuning workloads generated by OLTPBench and SYSBENCH. For fairness, we deactivate the optimizer recommendation module and adopt SMAC as the optimizer, consistent with LlamaTune. The order of issuing tuning tasks is randomized. Upon completing a tuning task, OpAdviser and Ottertune add the observation to their respective data repositories. Figure 13 illustrates the results. In the absence of historical observations, OpAdviser constructs the search space by extracting the effective region based on the target observations, when the target concordant ordering ratio exceeds 0.5 (better than random guessing). This mechanism demonstrates performance comparable to Hunter's knob selection and LlamaTune's randomized embeddings, both of which also reduce the search space based on target observations. Furthermore, OpAdviser's advantages over other baselines gradually become evident after completing the third tuning task, with the assistance of historical observations.

8 CONCLUSION

In this paper, we studied the techniques to enhance the efficiency of database tuning. Instead of focusing on designing advanced search optimizers, we concentrated on automating the construction of compact search space and selecting appropriate search optimizers for a given tuning task in an efficient manner. Our approach utilizes tuning history from past tasks to make informed decisions. Extensive experiments have shown that OpAdviser can significantly improve database tuning compared to state-of-the-art systems.

ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China (NSFC)(No. 61832001, U22B2037), Alibaba Group through Alibaba Innovative Research Program and National Key Research. Bin Cui and Yang Li are the corresponding authors.

REFERENCES

[1] 2015. TPC-H benchmark. http://www.tpc.org/tpch/.

[2] 2022. InnoDB Startup Options and System Variables. https://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html.

[3] 2022. Server System Variables. https://dev.mysql.com/doc/refman/5.7/en/server-system-variables.html.

[4] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. 2004. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB*. Morgan Kaufmann, 1110–1121.

[5] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD Conference*. ACM, 1009–1024.

[6] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Billian, and Andrew Pavlo. 2021. An Inquiry into Machine Learning-based Automatic Configuration Tuning Services on Real-World Database Management Systems. *Proc. VLDB Endow.* 14, 7 (2021), 1241–1253.

[7] Tianyi Bai, Yang Li, Yu Shen, Xinyi Zhang, Wentao Zhang, and Bin Cui. 2023. Transfer Learning for Bayesian Optimization: A Survey. *CoRR* abs/2302.05927 (2023).

[8] Christopher JC Burges. 2010. From ranknet to lambdarank to lambdamart: An overview. *Learning* 11, 23-581 (2010), 81.

[9] 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 *SIGMOD Conference*. ACM, 646–659.

[10] Stefano Cereda, Stefano Valladares, Paolo Cremonesi, and Stefano Doni. 2021. CGPTuner: a Contextual Gaussian Process Bandit Approach for the Automatic Tuning of IT Configurations Under Varying Workload Conditions. *Proc. VLDB Endow.* 14, 8 (2021), 1401–1413.

[11] Surajit Chaudhuri and Gerhard Weikum. 2006. Foundations of Automated Database Tuning. In *VLDB*. ACM, 1265.

[12] Mansheng Chen, Jia-Qi Lin, Xiang-Long Li, Bao-Yu Liu, Chang-Dong Wang, Dong Huang, and Jian-Huang Lai. 2022. Representation Learning in Multi-view Clustering: A Literature Review. *Data Sci. Eng.* 7, 3 (2022), 225–241.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT (1)*. Association for Computational Linguistics, 4171–4186.

[14] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (2013), 277–288.

[15] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *Proc. VLDB Endow.* 2, 1 (2009), 1246–1257.

[16] Yubin Duan, Ning Wang, and Jie Wu. 2022. Accelerating DAG-Style Job Execution via Optimizing Resource Pipeline Scheduling. *J. Comput. Sci. Technol.* 37, 4 (2022), 852–868.

[17] Ayat Fekry, Lucian Carata, Thomas F. J.-M. Pasquier, Andrew Rice, and Andy Hopper. 2020. To Tune or Not to Tune?: In Search of Optimal Configurations for Data Analytics. In *KDD*. ACM, 2494–2504.

[18] Ralf Herbrich, Thore Graepel, and Klaus Obermayer. 1999. Support vector learning for ordinal regression. (1999).

[19] Chun Kit Jeffery Hou and Kamran Behdinan. 2022. Dimensionality Reduction in Surrogate Modeling: A Review of Combined Methods. *Data Sci. Eng.* 7, 4 (2022), 402–427.

[20] Shiyue Huang, Yanzhao Qin, Xinyi Zhang, Yaofeng Tu, Zhongliang Li, and Bin Cui. 2023. Survey on performance optimization for database systems. *Sci. China Inf. Sci.* 66, 2 (2023).

[21] Shiyue Huang, Ziwei Wang, Xinyi Zhang, Yaofeng Tu, Zhongliang Li, and Bin Cui. 2023. DBPA: A Benchmark for Transactional Database Performance Anomalies. *Proc. ACM Manag. Data* 1, 1 (2023), 72:1–72:26.

[22] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *LION (Lecture Notes in Computer Science)*, Vol. 6683. Springer, 507–523.

[23] Konstantinos Kanellis, Ramnatthan Alagappan, and Shivaram Venkataraman. 2020. Too Many Knobs to Tune? Towards Faster Database Tuning by Pre-selecting Important Knobs. In *HotStorage*. USENIX Association.

[24] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. 2022. LlamaTune: Sample-Efficient DBMS Configuration Tuning. *Proc. VLDB Endow.* 15, 11 (2022), 2953–2965.

[25] Aaron Klein. 2017. RoBO : A Flexible and Robust Bayesian Optimization Framework in Python.

[26] Jan Kossmann and Rainer Schlosser. 2020. Self-driving database systems: a conceptual approach. *Distributed Parallel Databases* 38, 4 (2020), 795–817.

[27] Mayuresh Kunjir and Shivnath Babu. 2020. Black or White? How to Develop an AutoTuner for Memory-based Analytics. In *SIGMOD Conference*. ACM, 1667–1683.

[28] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.

[29] Stefan Lessmann, Robert Stahlbock, and Sven F Crone. 2005. Optimizing hyper-parameters of support vector machines by genetic algorithms.. In *IC-AI*. 74–82.

[30] David D. Lewis and Jason Catlett. 1994. Heterogeneous Uncertainty Sampling for Supervised Learning. In *ICML*. Morgan Kaufmann, 148–156.

[31] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* 12, 12 (2019), 2118–2130.

[32] Yang Li, Huaijun Jiang, Yu Shen, Yide Fang, Xiaofeng Yang, Danqing Huang, Xinyi Zhang, Wentao Zhang, Ce Zhang, Peng Chen, and Bin Cui. 2023. Towards General and Efficient Online Tuning for Spark. *Proc. VLDB Endow.* 16, 12 (2023), 3570–3583.

[33] Jinqing Lian, Xinyi Zhang, Yingxia Shao, Zenglin Pu, Qingfeng Xiang, Yawen Li, and Bin Cui. 2023. ContTune: Continuous Tuning by Conservative Bayesian Optimization for Distributed Stream Data Processing Systems. *CoRR* abs/2309.12239 (2023).

[34] Hao Liao, Qi-Xin Liu, Ze-cheng Huang, Ke-Zhong Lu, Chi Ho Yeung, and Yi-Cheng Zhang. 2022. Accumulative Time Based Ranking Method to Reputation Evaluation in Information Networks. *J. Comput. Sci. Technol.* 37, 4 (2022), 960–974.

[35] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. Continuous control with deep reinforcement learning. In *ICLR (Poster)*.

[36] Scott M. Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *NIPS*. 4765–4774.

[37] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *SIGMOD Conference*. ACM, 631–645.

[38] Michael D. McKay. 1992. Latin Hypercube Sampling as a Tool in Uncertainty Analysis of Computer Models. In *WSC*. ACM Press, 557–564.

[39] Amin Nayebi, Alexander Munteanu, and Matthias Poloczek. 2019. A Framework for Bayesian Optimization in Embedded Subspaces. In *ICML (Proceedings of Machine Learning Research)*, Vol. 97. PMLR, 4752–4761.

[40] Valerio Perrone and Huibin Shen. 2019. Learning search spaces for Bayesian optimization: Another view of hyperparameter transfer learning. In *NeurIPS*. 12751–12761.

[41] Omer Sagi and Lior Rokach. 2018. Ensemble learning: A survey. *WIREs Data Mining Knowl. Discov.* 8, 4 (2018).

[42] Dennis E. Shasha and Philippe Bonnet. 2002. Database Tuning: Principles, Experiments, and Troubleshooting Techniques. In *VLDB*. Morgan Kaufmann.

[43] Dennis E. Shasha and Steve Rozen. 1992. Database Tuning. In *VLDB*. Morgan Kaufmann, 313.

[44] Adam J. Storm, Christian Garcia-Arellano, Sam Lightstone, Yixin Diao, and Maheswaran Surendra. 2006. Adaptive Self-tuning Memory in DB2. In *VLDB*. ACM, 1081–1092.

[45] Immanuel Trummer. 2022. DB-BERT: A Database Tuning Tool that "Reads the Manual". In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 190–203. https://doi.org/10.1145/3514221.3517843

[46] Bing Wei, Limin Xiao, Yao Song, Guangjun Qin, Jinbin Zhu, Baicheng Yan, Chaobo Wang, and Zhisheng Huo. 2022. A self-tuning client-side metadata prefetching scheme for wide area network file systems. *Sci. China Inf. Sci.* 65, 3 (2022).

[47] Gerhard Weikum, Axel Mönkeberg, Christof Hasse, and Peter Zabback. 2002. Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. In *VLDB*. Morgan Kaufmann, 20–31.

[48] David H. Wolpert and William G. Macready. 1997. No free lunch theorems for optimization. *IEEE Trans. Evol. Comput.* 1, 1 (1997), 67–82. https://doi.org/10.1109/4235.585893

[49] Huan Zhang, Liangxiao Jiang, and Chaoqun Li. 2022. Attribute augmented and weighted naive Bayes. *Sci. China Inf. Sci.* 65, 12 (2022).

[50] 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 *SIGMOD Conference*. ACM, 415–432.

[51] 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. *Proc. VLDB Endow.* 15, 9 (2022), 1808–1821.

[52] Xinyi Zhang, Zhuo Chang, Hong Wu, Yang Li, Jia Chen, Jian Tan, Feifei Li, and Bin Cui. 2023. A Unified and Efficient Coordinating Framework for Autonomous DBMS Tuning. *Proc. ACM Manag. Data* 1, 2 (2023), 186:1–186:26.

[53] 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 *SIGMOD Conference*. ACM, 2102–2114.

[54] Xinyi Zhang, Hong Wu, Yang Li, Jian Tan, Feifei Li, and Bin Cui. 2022. Towards Dynamic and Safe Configuration Tuning for Cloud Databases. *CoRR*

abs/2203.14473 (2022).

[55] Xinyi Zhang, Hong Wu, Yang Li, Jian Tan, Feifei Li, and Bin Cui. 2022. Towards Dynamic and Safe Configuration Tuning for Cloud Databases. In *SIGMOD Conference*. ACM, 631–645.

[56] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. BestConfig: tapping the performance potential of systems via automatic configuration tuning. In *SoCC*. ACM, 338–350.

[57] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. 2021. A Comprehensive Survey on Transfer Learning. *Proc. IEEE* 109, 1 (2021), 43–76.