ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases

Xinyi Zhang[*†‡]
Peking University & Alibaba Group
zhang_xinyi@pku.edu.cn

Hong Wu[*‡]
Alibaba Group
hong.wu@alibaba-inc.com

Zhuo Chang[‡§]
Alibaba Group & Peking University
z.chang@pku.edu.cn

Shuowei Jin[‡]
Alibaba Group
shuowei.jsw@alibaba-inc.com

Jian Tan[‡]
Alibaba Group
j.tan@alibaba-inc.com

Feifei Li[‡]
Alibaba Group
lifeifei@alibaba-inc.com

Tieying Zhang[‡]
Alibaba Group
tieying.zhang@alibaba-inc.com

Bin Cui[†§¶]
Peking University
bin.cui@pku.edu.cn

ABSTRACT

Modern database management systems (DBMS) contain tens to hundreds of critical performance tuning knobs that determine the system runtime behaviors. To reduce the total cost of ownership, cloud database providers put in drastic effort to automatically optimize the resource utilization by tuning these knobs. There are two challenges. First, the tuning system should always abide by the service level agreement (SLA) while optimizing the resource utilization, which imposes strict constrains on the tuning process. Second, the tuning time should be reasonably acceptable since time-consuming tuning is not practical for production and online troubleshooting.

In this paper, we design ResTune to automatically optimize the resource utilization without violating SLA constraints on the throughput and latency requirements. ResTune leverages the tuning experience from the history tasks and transfers the accumulated knowledge to accelerate the tuning process of the new tasks. The prior knowledge is represented from historical tuning tasks through an ensemble model. The model learns the similarity between the historical workloads and the target, which significantly reduces the tuning time by a meta-learning based approach. ResTune can efficiently handle different workloads and various hardware environments. We perform evaluations using benchmarks and real world workloads on different types of resources. The results show that, compared with the manually tuned configurations, ResTune reduces 65%, 87%, 39% of CPU utilization, I/O and memory on average, respectively. Compared with the state-of-the-art methods, ResTune finds better configurations with up to ~ 18× speedups.

CCS CONCEPTS

• **Information systems** → **Autonomous database administration**; • **Computing methodologies** → **Machine learning**.

KEYWORDS

resource; tuning; cloud database; service level agreement

ACM Reference Format:
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 (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457291

1 INTRODUCTION

Tuning configuration knobs of modern database management systems (DBMS) is critical for system performance, albeit challenging. Different knobs directly affect the running database performance and jointly determine the quality of service and the resource utilization of DBMS. As a common practice, to apply an appropriate configuration for a given workload, database administrators (DBAs) are responsible for tuning these knobs based on experience. However, in a cloud environment, manually tuning possibly tens to hundreds of controlling knobs do not guarantee the performance across various workloads and could not scale. Therefore, automatic tuning becomes an appealing feature for cloud providers.

On one hand, optimizing the system performance (e.g., throughput, latency) is critical to improving users' experience. On the other hand, controlling the resource utilization is a necessity from the cloud provider's perspective, due to the following reasons. First, one of the goals of using cloud databases is to reduce the Total Cost of Ownership (TCO). Maintaining a low cost is an important economic factor to attract users, which urges to more efficiently utilize the available computing resources. Second, optimizing computing resources such as CPU, memory, and I/O helps troubleshoot performance bugs that cause unnecessary high utilization. High resource

[*]Xinyi Zhang and Hong Wu contribute equally to this paper.
[†]Center for Data Science, Peking University & National Engineering Laboratory for Big Data Analysis and Applications
[‡]Database and Storage Laboratory, Damo Academy, Alibaba Group
[§]School of EECS & Key Laboratory of High Confidence Software Technologies, Peking University
[¶]Institute of Computational Social Science, Peking University (Qingdao)

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457291

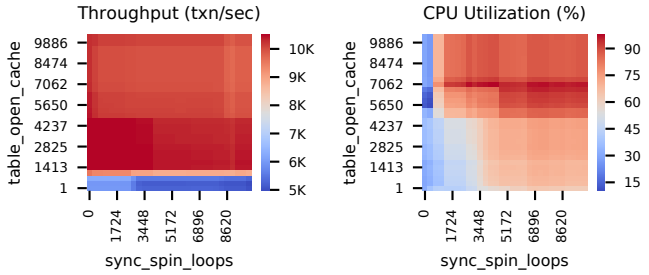

Figure 1: TPS and CPU Usage for Real Workload with 2 Knobs

utilization often leads to unpredictable system hangs and resource contentions in a shared or multi-tenant environment [9, 20]. For example, high CPU utilization is a frequent issue that affects the availability of cloud databases [2]. Third, the throughput of real workloads is often bounded by the request rate determined by the clients. Thus, the request rates do not necessarily reach the processing capacity of DBMS. For these common application scenarios, squeezing more throughput from the capacity is not the goal. Meanwhile, controlling resource utilization is more valuable for end-users, which can help them to choose appropriate cloud instance types and to further avoid over-provisioning.

One challenge of tuning configuration knobs is to reduce resource utilization while still guaranteeing the Service Level Agreement (SLA), e.g., without violating the throughput and latency requirements. Figure 1 plots the throughput along with CPU usage on a real workload with 2 controlling knobs, i.e., the number of open tables[1] and the number of times a thread waits for the mutex to be freed before suspending[2]. The result shows that, even though a wide range of configurations has different CPU usages, they experience the same throughput. As mentioned earlier, the throughput of real workloads is often bounded by the user request rate. Therefore, there are opportunities to optimize resource utilization without sacrificing the SLA. Most existing database tuning methods [11, 19, 27, 49] mainly focus on improving the throughput and latency without optimizing the resource usage and SLA simultaneously. For example, iTuned [11] and OtterTune [6] use Gaussian Processes to tune knobs to achieve only high throughputs. CDBTune [49] and QTune [27] use the reinforcement learning approach to train a policy model to recommend good knobs, which, however, takes a long time to learn the model [23].

The other challenge is to satisfy the natural constraint imposed by the real applications that often limit the required tuning times. Tuning systems replay the workload repeatedly to learn the model iteratively, and the replay times dominate the tuning process. The state-of-the-art systems [6, 49] take hundreds to thousands of iterations to find an ideal configuration. For typical benchmarks that assume the transaction statistics do not change over time, the replay time can be set to 3-5 minutes [49]. But for real workloads, we observe that the replay time for each iteration takes at least 5 minutes to adapt to different types of transactions. This could cause the total tuning time for real workloads to last for a few days. This issue is more pronounced when considering that tuning itself requires computing resources such as DBMS copies to replay on the user side (Section 4). Thus, the tuning time should be minimized. In addition, tuning DBMS systems, e.g., reducing the high resource utilization, can be used for online performance troubleshooting. High utilization could have a severe impact on system availability. From this point of view, the tuning time should match the typical system recovery time, which is often from a few minutes to 1 hour [1]. To accelerate the tuning process by reducing the budget to tens of iterations, ResTune utilizes the historical data collected from tuning other tasks and transfer the experience into tuning new tasks. This requires the tuning algorithm to efficiently and effectively represent useful knowledge from historical tuning data.

Our Approach. Different from previous works that only consider the throughput and latency, in this paper, we define the *resource-oriented tuning problem* that aims to find the configurations to minimize the resource usage without sacrificing the throughput and latency. We formulate it as a constrained optimization problem and propose ResTune, a constraint-aware database tuning system boosted by meta-learning. ResTune is a tool provided by the cloud providers, which aims to reduce the Total Cost of Ownership for its end users. It optimizes the resource utilization for a given workload by imposing constraints on the performance requirements. ResTune models both the objective function and the constraints using Gaussian processes to recommend configurations with optimized resource utilization while guaranteeing the SLA. To improve the efficiency of ResTune, we use meta-learning, which is the method of systematically learning from meta-data to accomplish new tasks [44]. A novel meta-learning pipeline is proposed to use multiple models (base-learners) to represent prior knowledge and an ensemble model (meta-learner) to combine and effectively utilize the experiences. The meta-learner measures the usefulness of base-learners to target workload through meta-feature and model prediction. In this way, ResTune could accordingly make use of existing data and accelerate the tuning process. Furthermore, our approach can transfer the knowledge over different workloads and heterogeneous hardware environments.

Specifically, we make the following contributions:

- To deal with the challenges in real DBMS scenarios, we formulate the resource-oriented configuration tuning problem as a constrained Bayesian Optimization problem.
- To accelerate the tuning process within an acceptable time interval, a meta-learning strategy is proposed to extract experience from past tasks. Unlike previous studies, our approach uses relative rankings rather than absolute distances to measure the similarity between workloads. It can better transfer knowledge across different hardware environments and achieve fast tuning and efficient adaptation. To the best of our knowledge, this is the first attempt to boost constrained Bayesian Optimization with meta-learning for tuning DBMS.
- We implement the proposed method and evaluate on standard benchmarks and real workloads. Compared with the manual configurations provided by the DBAs, ResTune reduces 65% of CPU utilization, 87% of I/O, and 39% of memory on average. Compared with the state-of-the-art DBMS tuning systems, ResTune finds better configurations with up to $\sim$ 18$\times$ speedups.

The remainder of the paper is organized as follows. Section 2 provides the related work and Section 3 formally defines the

[1]MySQL knob: table_open_cache
[2]MySQL knob: innodb_sync_spin_loops

resource-oriented tuning problem. An overview of our approach is given in Section 4, followed by solving the constrained optimization problem in Section 5. We propose the meta-learning approach to accelerate the tuning process in Section 6. Section 7 presents the experimental results. Last, we conclude in Section 8.

2 RELATED WORK

Tuning Knobs. There has been an active area of research on tuning configuration knobs of DBMS recently. Many works [6, 11, 27, 49] have studied auto-tuning the knobs, but they mainly focus on optimizing the performance. Although they do not consider reducing the resource usage, the methods are valuable, which can be mainly classified into three categories: search-based, Bayesian Optimization (BO) [37] based and Reinforcement Learning (RL) [28] based.

- **Search-based.** BestConfig [51] is a search-based method, which tries to find good configuration according to several heuristics. Whenever a new tuning request comes, they will restart the entire search process and not take advantage of past experience.
- **BO-based.** Ottertune [6] and iTuned [11] uses the BO-based method, modeling the tuning as a black-box optimization problem. Ottertune also considers past experience by using a workload mapping strategy. However, this strategy can not adapt to the changes to hardware [6], which restrains using extensive data in the cloud environment. Another work [35] uses the Gaussian process to predict response times of queries to meet the SLAs, while we focus on optimizing resource usage by tuning knobs.
- **RL-based.** RL methods in [23, 27, 49] tunes the performance of DBMS by learning a neural network between the internal metrics and the configuration knobs. Its training overload is relatively high, so it takes thousands of iterations to train SYSBENCH [27].

Bayesian Optimization with Constraints. In realistic scenarios, it is often necessary to satisfy a few constraints, such as memory consumption [36, 41], prediction time [14, 15].The simplest way to solve constrained optimization is to define a penalty value and attach it to the objective function when violating the constraints [12, 18, 41]. More advanced approaches model the possibility of violating one or more constraints and search for configurations that are unlikely to violate any constraints [14, 24]. These works inspire us to solve the resource-oriented tuning problem.

Meta-Learning for HPO. Recently, works on Hyper-Parameters Optimize (HPO) utilize meta-learning to borrow strength from meta-data [8, 16, 32, 33, 45–47, 50]. Meta-data are the data describing previous learning tasks and learned models, including measurable properties of the task, also known as meta-features [43]. Our design is highly motivated by RGPE [13]. It trains a weighted surrogate on each history and target task to optimize machine learning hyper-parameters. We adopt its ensemble idea and apply it to the resource-oriented problem. To the best of our knowledge, this is the first work to boost constrained Bayesian Optimization with meta-learning. To evaluate feasibility across tasks, we define constraints in a unified scale and propose two weight assignment strategies.

3 PROBLEM STATEMENT

We formalize the resource-oriented tuning problem as an optimization problem with SLA constraints. The objective is to minimize resource usage without violating the constraints on the throughput and latency requirements. The cloud database providers guarantee the SLA [38][39] under the default configuration. Therefore, the constraints are determined by the throughput and latency under default configuration before the resource-oriented tuning starts. ResTune assures the database performance does not downgrade from the previous configuration (e.g., the DBA default). Sometimes slow queries or other anomalies can break the SLA, but these are issues orthogonal to our tuning tasks.

Constrained Optimization Problem. Consider a database system with a continuous configuration space $\Theta = \Theta_1 \times \Theta_2 \times ... \times \Theta_m$ with $\Theta_i \in [0, 1]$. We normalize the range of the knobs into [0,1]. For knobs taking discrete values, we first partition [0, 1] into bins and then round each value to the nearest bin after normalization. Let f_{res}, f_{tps}, f_{lat} denote the resource utilization (e.g., f_{cpu}, f_{memory} or f_{io}), throughput and 99%th percentile latency, respectively. For a given workload, we want to find the configuration $\theta^* \in \Theta$ that minimizes resource usage and satisfies the SLA requirements. The resource-oriented tuning problem is defined as follows:

$$\begin{aligned} &\arg\min_{\theta} f_{res}(\theta), \\ \text{s.t.} \quad &f_{tps}(\theta) \geq \lambda_{tps} \\ &f_{lat}(\theta) \leq \lambda_{lat} \end{aligned} \tag{1}$$

with λ_{tps} and λ_{lat} being the lower bound of the throughput and the upper bound of the latency, respectively.

4 OVERVIEW OF RESTUNE

Figure 2 presents the overview workflow of ResTune. When a tuning task is launched, a copy instance of the target DBMS is initiated, and a time window of the target workload is collected for future replay. The copy instance is deployed in the user's environment, e.g., dedicated Virtual Private Cloud, to protect data privacy. Two main parts of ResTune are deployed separately, ResTune Client and ResTune Server. ResTune Client consists of the Meta-Data Processing component and the Target Workload Replay component, and it is deployed in the user's environment. It handles the pre-processing of the target task and the evaluation of suggested knobs. ResTune Server is responsible for recommending the configuration for each iteration, and it is deployed in the backend tuning cluster, consisting of two components: Knowledge Extraction and Knobs Recommendation. After pre-processing the target task in ResTune Client, the meta-feature and the base model of the target task are sent to the Knowledge Extraction component of ResTune Server. ResTune Server then calculates the static weights and the dynamic weights based on the target task's input, meta-features and base models stored in the data repository. After ensembling, a meta-learner is used to recommend new configurations. Finally, the knobs are applied to the database, and the replayer is triggered. Then, the evaluation results are appended in the observations of the target task, finishing an iteration. When the tuning task ends, the meta-data of the task is collected to the data repository.

Data Repository. Data Repository maintains meta-features and base models generated from previous tuning tasks on various workloads and hardware environments. The meta-feature is an embedding vector representing the workload's overall resource utilization,

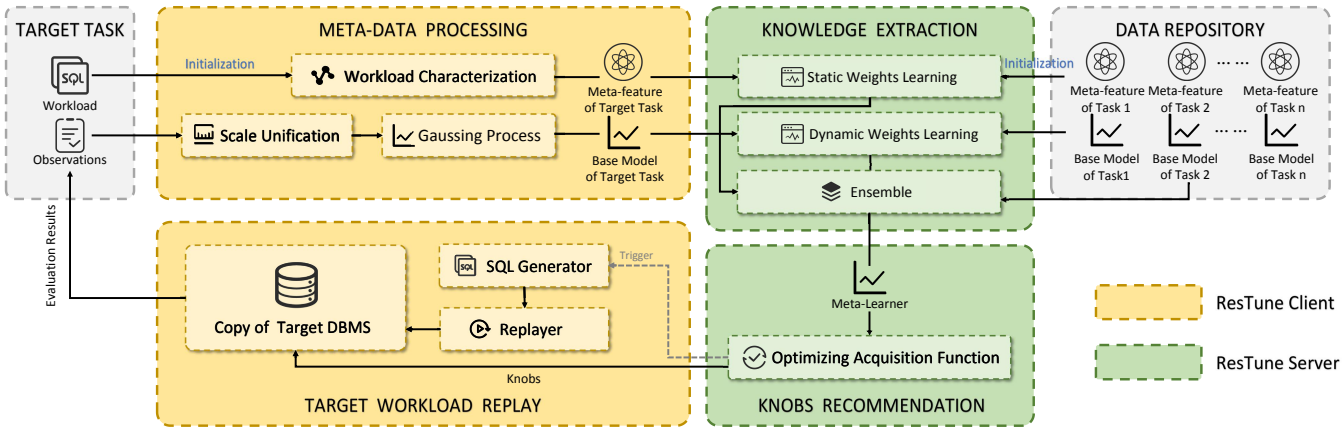

Figure 2: Overall Architecture of ResTune

and the base model is a multi-output Gaussian Processing model fitted with historical observations on resource utilization, throughput, and latency (defined in Section 5.1).

Meta-Data Processing. With the target workload and its observations, the Meta-Data Processing component calculates the meta-feature and generates a base-learner for the target task. The meta-feature is calculated through the workload characterization pipeline. At initialization, when observations are insufficient to fit an accurate model, meta-feature can serve as a general understanding of the target tuning task. When generating base-learners, there exist various hardware instances in cloud databases, and their scales of objective function differ a lot. Therefore, ResTune first unifies the meta-data to the consistent scale by standardizing the observations to have zero mean and unit standard deviation. The values of constraints on throughput and latency are also re-scaled, forming a constrained optimization task ready for further learning.

Knowledge Extraction. The Knowledge Extraction component uses the method proposed in Section 6 to combine the base-learners into a meta-learner in a weighted ensemble manner. It adopts two strategies to assign the weights: *static* and *dynamic* weights learning. At initialization, the *static weights learning* assigns weights based on the meta-feature distances. Afterward, ResTune uses the *dynamic weights learning* strategy, comparing the relative ranking between the prediction from the base-learner and the target task's observations. With the dynamic assigning strategy, the weights update as the number of observations for the target workload increases (Section 6.4.2). With these two strategies, we get a meta-learner that serves as a learned surrogate model boosted with experience.

Knobs Recommendation. After learning the weights, the learned surrogate model is ready to predict the performance and resource utilization on new configurations. With the predictability of the meta-learner, ResTune generates promising configurations of knobs that satisfy the re-scaled constraints. Internally, the Knobs Recommendation component suggests the next configuration to evaluate by optimizing the constrained acquisition function (Section 5).

Target Workload Replay. Once the recommended configuration applies to the database copy, the workload generator triggers to replay the user's workload under the same environment. Replaying the same query repeatedly would cause the write operations (INSERT, UPDATE, DELETE) invalid due to the conflict of primary keys or foreign keys. To solve this, the Target Workload Replay component extracts the query template from the workload and sample the scalar value and variable name before replaying. Furthermore, to reproduce the user's real behavior, the replayer supports executing the queries at a given request rate. In our experiment, we use the same request rate as the target workload. After replaying, the evaluation results of resource utilization, throughput, latency are collected in the observation data. The tuning process keeps iterating until the decline in resource utilization reaches the goal or the tuning model is converged. If changes in resource utilization, throughput and latency do not exceed 0.5% in 10 consecutive iterations, the model is considered to be converged.

5 SOLVING CONSTRAINED OPTIMIZATION

As we discussed, resource-oriented tuning can be formalized as a constrained optimization problem. In this section, we introduce Constrained Bayesian Optimization (CBO), which extends Bayesian Optimization with an inequality-constrained setting. ResTune could therefore recommend configurations with optimized resource utilization while guaranteeing the SLA (Service Level Agreement).

5.1 Modeling Constrained Functions

In our resource-oriented tuning problem, constraint functions $f_{tps}(\theta)$ and $f_{lat}(\theta)$ can be observed simultaneously with the objective function $f_{res}(\theta)$. But they are also expensive-to-evaluate black-box functions the same as $f_{res}(\theta)$. We use Gaussian Process [31], a powerful learning technique with power equivalent to that of deep networks [6]. It allows us to approximate complex response surfaces through adaptively sampling of the search space in a manner balancing exploration and exploitation. GP outputs the estimated confidence bound on the predictions, supports noisy observations, and has the ability to use gradient-based methods [34]. In ResTune, we model $f_{tps}(\theta)$ and $f_{lat}(\theta)$ with a conditionally independent Gaussian Process, denoted by $\tilde{f}_{tps}(\theta)$ and $\tilde{f}_{lat}(\theta)$. Similarly, probabilistic model of the resource utilization is denoted by $\tilde{f}_{res}(\theta)$.

During the optimization process, we keep a data set H that records the historical observations in the form of four-tuples: $H =$

$\left\{\left(\theta_i, f_{res}\left(\theta_i\right), f_{tps}\left(\theta_i\right), f_{lat}\left(\theta_i\right)\right)\right\}_{i=1}^{n}$, corresponding to the knobs configuration and the value of resource utilization, throughput and latency under this configuration. We denote configuration that satisfies the performance constraints as feasible configuration. Note that H includes both the data of feasible configurations and infeasible ones. Although infeasible configurations are not considered to be recommended as final results, they are useful to the optimization process for two factors. First, infeasible configurations can indicate which regions of the configuration space are more likely to be feasible, which reduces the exploration to the substandard region. Second, infeasible configurations help determine the shape and descent directions of the objective function and the constrained functions, contributing to more efficient exploitation.

Based on the observation data H, we maintain three independent Gaussian Processing models with mean $\mu_u(\theta)$ and variance $\sigma_u^2(\theta)$, $u \in \{res, tps, lat\}$. After each iteration, we add the latest observation data in H and update the three GP models. They can be formalized as a multi-output Gaussian Process model that outputs the predictions with confidence bounds.

5.2 Guiding Search in Feasible Region

To facilitate the recommendation of feasible configurations, we reconsider the acquisition function in basic Bayesian Optimization. The new acquisition function should guide the selection of configurations with less resource utilization in feasible areas.

We denote the improvement function of a candidate point θ in terms of the objective function as $I(\theta)$ and its expected form as $\alpha_{EI}(\theta)$. The Expected Improvement $\alpha_{EI}(\theta)$ is a state-of-the-art acquisition function in BO since it be calculated in closed form. The $\alpha_{EI}(\theta)$ is defined below:

$$\alpha_{EI}(\theta) = \mathbb{E}\left[max(0, f_{res}(\theta_{best}) - f_{res}(\theta)\right] \tag{2}$$

However, directly applying the EI function would lead to infeasible knobs recommendation with low resource utilization at the cost of database performance. To solve our constrained optimization problem, we extend the acquisition function with two modifications. First, different from the original definition in the EI function, θ_{best} is redefined as the best *feasible* point, meaning the configuration has the lowest resource utilization and satisfying the throughput and latency constraints. Second, improvement of the infeasible points is assigned a value of zero, meaning infeasible points achieve no improvement. Now we define the improvement function with constraint for a candidate θ:

$$I_C(\theta) = \Delta(\theta) max(0, f_{res}(\theta_{best}) - f_{res}(\theta)) \tag{3}$$

The feasible indicator function $\Delta(\theta)$ is one when θ is feasible or zero, otherwise. Using the probabilistic surrogate model, the improvement function with constraints can be calculated as:

$$\tilde{I}_C(\theta) = \tilde{\Delta}(\theta) max(0, f_{res}(\theta_{best}) - \tilde{f}_{res}(\theta)) = \tilde{\Delta}(\theta)\tilde{I}(\theta) \tag{4}$$

The term $\tilde{\Delta}(\theta)$ is a Bernoulli random variable with expectation $\mathbb{E}[\tilde{\Delta}(\theta)] = Pr[\tilde{f}_{tps}(\theta) \geq \lambda_{tps}, \tilde{f}_{lat}(\theta) \leq \lambda_{lat}]$. Note that $Pr[f_{tps}(\theta) \geq \lambda_{tps}, f_{lat}(\theta) \leq \lambda_{lat}]$ is a multivariate Gaussian probability. Considering the independence, it can be factorized as $Pr[f_{tps}(\theta) \geq \lambda_{tps}] \cdot Pr[f_{lat}(\theta) \leq \lambda_{lat}]$, a product of two Gaussian cumulative distribution functions. Finally, the constrained acquisition function can be constructed, we call it *Constrained Expected Improvement*(CEI):

$$\begin{aligned} \alpha_{CEI}(\theta) &= \mathbb{E}\left[\tilde{I}_C(\theta)|\theta\right] = \mathbb{E}\left[\tilde{\Delta}(\theta)\tilde{I}(\theta)|\theta\right] = \mathbb{E}\left[\tilde{\Delta}(\theta)|\theta\right]\mathbb{E}\left[\tilde{I}(\theta)|\theta\right] \\ &= Pr[\tilde{f}_{tps}(\theta) \geq \lambda_{tps}] \cdot Pr[\tilde{f}_{lat}(\theta) \leq \lambda_{lat}] \cdot \alpha_{EI}(\theta) \end{aligned} \tag{5}$$

According to the definition, the acquisition function $\alpha_{CEI}(\theta)$ is the EI function of θ over the best feasible point so far, weighted by the probability that both $f_{tps}(\theta)$ and $f_{lat}(\theta)$ satisfy the constrains. The CEI function can well balance the expected objective value with the probability of feasibility. Therefore, ResTune can recommend promising feasible configurations.

6 BOOSTING TUNING PROCESS

With the constrained Bayesian Optimization technique, ResTune can solve the resource-oriented tuning problem. However, different target workloads are tuned independently, without considering the correlation among them. Intuitively, the same workloads running on different hardware share information for tuning knobs. Even for different tasks, the relationship between hidden features can lead to knowledge sharing. As cloud providers, we can collect abundant tuning data from numerous tasks and further accelerate the tuning process of new tasks. Motivated by meta-learning, we propose a framework that combines base-learners and generates a meta-learner L_M for the target workload. Base-learners are learning subsystems adaptive with the experience [25]. With a set of prior observations and a small set of known observations on a new task (target task), a meta-learner is a learned model that can suggest good configurations for the new task. In Section 6.1 and 6.2 describe what are the knowledge ResTune use. Section 6.3 and 6.4 describe how to combine the base-learners for tuning the target task.

6.1 Scale Unification

There are various of instances and workloads in could database, making the scales of metrics among tuning tasks differ a lot. In Section 5, each tuning task is independent, forming the observation track $\{\theta_j, f_u(\theta_i)\}_{j=1}^{n}, u \in \{\text{throughput, latency, resource utilization}\}$. We denote the observation history for tuning task w_i by $\mathcal{H}_i = \{\theta_j^i, f_u(\theta_j^i, w_i)\}_{j=0}^{n_i}$, $i = 1, ..., T$. T is the total number of historical tasks. To further utilize historical observations to evaluate configuration's performance and feasibility, we need to unify the metrics of different scales. We adopt a simple strategy that standardizes the observations for each task separately to have zero mean and unit standard deviation [48]. As a result, the output prediction of the base-learner is a relative value instead of absolute performance $f(\theta, w_i)$. Meta-learner also outputs a relative value, denoted by $L_M(\theta)$. Accordingly, the constraint $f_u(\theta) \leq \lambda_u, u \in \{\text{tps, lat}\}$ needs to be transformed to $L_M^u(\theta) \leq \lambda_u^{'}$ with a re-scaled value $\lambda_u^{'}$. Since we usually use the performance metrics of default configuration as constraints, i.e. $\lambda_u = f_u(\theta_d, w_{target})$, where θ_d is the default configuration. We can set re-scaled $\lambda_u^{'}$ as $L_M^u(\theta_d)$.

Proof. If meta-learner predicts the relative performance value of θ to be a smaller value than that of θ_d .i.e. $L_M^u(\theta) \leq L_M^u(\theta_d)$, then it's predicted that $f_u(\theta, w_t) \leq f_u(\theta_d, w_t)$ .i.e $f_u(\theta, w_t) \leq \lambda_u$. If meta-learner predicts the the relative performance value of θ to be a larger value than that of θ_d .i.e. $L_M^u(\theta) \geq L_M^u(\theta_d)$, then it's

predicted that $f_u(\theta, w_t) \geq f_u(\theta_d, w_t)$ .i.e $f_u(\theta, w_t) \geq \lambda_u$. There the re-scaled constraint $\lambda_u^{'}$ can be set as $L_M^u(\theta_d)$.

6.2 Workload Characterization

Another kind of meta-data is characterizations of the workload. At initialization, the observations of the target task are insufficient. Therefore, ResTune uses meta-features to find similar base-learners. We propose a workload characterization pipeline to extract the meta-feature vector m_i for each tuning task. Then the weights of base-learners are calculated based on the similarity measured on the meta-feature. The pipeline of workload characterization is discussed as follows, it produces the meta-feature that represents the estimated resource cost of the workload.

Feature Extraction. Different workloads have different queries containing specific variables. The challenge is that our model for meta-feature should generalize over different database schemas and SQL queries. As we discussed, the workload characterization component is deployed in the user's environment, simple models are preferred. Although using statistics from optimizer such as plan structures are beneficial, it needs hand-engineered feature engineering effort. Instead, we only use the SQL queries without collection additional information during execution. However, the variable names and digits used in SQL queries are unbounded, which makes generalization across workloads and schemas difficult [21, 52]. Note that the query templates exhibit only a small number of different patterns, it allows us to adopt the TF-IDF (term frequency-inverse document frequency) method [17]. Specifically, each query contains reserved SQL keywords (e.g., SELECT, UPDATE, DISTINCT), and each reserved word represents a certain type of operation in DBMS. We extract the reserved words by filtering out the specific variables and calculate the TF-IDF feature vector for each query. Since only the reserved words are used, the vocabulary dictionary is small, and the model has better generality.

Classification Model. Based on the TF-IDF feature vectors, we adopt a random forest model to classify each query. To train the model, We use the resource cost level as labels, which have a wide range of values and are highly skewed. To prevent the models from being too sensitive to queries with a large label value (outliers), we apply a logarithmic transformation to the values of these labels[52][26] and discrete labels to do the classification.

Workload Embedding Procedure. For each workload, we identify the reserved words for each query and then apply the TF-IDF transformer. The TF-IDF feature vectors are used as the input to the random forest model for classification, which returns the predicted probability distribution. We compute the average of the probability distributions (as vectors) for all the queries across the whole input workload. The averaged probability distribution represents the meta-feature for the input workload by characterizing the appearance frequencies of the queries.

6.3 Knowledge Extraction

A simple way to extract the historical knowledge is to combine all past runs into the same model as observations in the current run. However, this approach has two drawbacks. First, it does not scale well with the number of past runs. Gaussian Process (GP) regression has $O(n^3)$ complexity, and combining t past runs with n iterations into a single model incurs $O(t^3n^3)$ computational cost. Second, learning a GP on all training instances relies on the strong assumption that each tuning task from different workloads or hardware environments is equally important [47]. Mapping to the most similar workload like [6] still can not solve this problem, as the target workload and the mapped workload may not be identical. Therefore, we need a scalable meta-learner that can distinguish experience and predict the output.

In our approach, base-learners l_i, $i = 1, ..., T$ are responsible for memorizing historical observations H_i. We also fit a model for the target workload w_{T+1}. As discussed, we use the Gaussian Process model to fit each tuning task's observations as a base-learner. So we have $T + 1$ base-learners with mean function $\mu_i(\theta)$, variance function $\sigma_i^2(\theta)$. Together with all the base-learners, we can train a meta-learner L_M to predict the performance of the configuration θ_i on target workload w_{T+1}. Since GP provides a theoretically justified way to trade off exploration (i.e., acquiring new knowledge) and exploitation (i.e., making decisions based on existing knowledge) [22], we still adopt GP as meta-leaner L_M, $L_M(\theta|H) \sim \mathcal{N}(\mu_M(\theta), \sigma_M^2(\theta))$. It extracts knowledge from base-learners in a weighted manner. The mean function is a weighted combination of the predictions of each base-learners:

$$\mu_M(\theta) = \frac{\sum_{i=1}^{T+1} g_i \mu_i(\theta)}{\sum_{i=1}^{T+1} g_i} \tag{6}$$

where g_i is the weight of base-learner l_i. Considering the observation from target workload should be more trusted, we define $\sigma_M^2(\theta)$ to rely on the uncertainty of target base-learner l_{T+1} only, where v_i is assigned to one for target base-learner and 0 otherwise:

$$\sigma_M^2(\theta) = \sum_{i=1}^{T+1} v_i \sigma_i^2(\theta), \qquad v_i = \begin{cases} 1 & \text{i=T+1} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

In this way, the complexity of tuning keeps $O(n^3)$, meaning combining meta-data does not add computational complexity.

6.4 Base-Learner Evaluation

How to combine the base-learners according to meta-data is vital to the performance of our meta-learning design. As we discussed in Section 6.3, we assign a weight for each base-learner. We propose two techniques to combine base-learners, which refer to static weight learning and dynamic weight learning.

6.4.1 Learning from meta-feature. At initialization, we measure the similarity for tuning tasks based on the meta-feature of workload. If a workload is more similar to the target workload, a larger weight is assigned to its base-learner. Therefore, the weight g_i of base-learner l_i is the similarity between the workload W_i's meta-feature m_i and the target workload's meta-feature m_{T+1}. We choose Epanechnikov quadratic kernel [30] as the measurement of similarity.

$$g_i = \gamma\left(\frac{\|m_i - m_{T+1}\|_2}{\rho}\right), \qquad \gamma = \begin{cases} \frac{3}{4}(1 - t^2) & t \leq 1 \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

and $\gamma \geq 0$ is the bandwidth. Consequently, meta-learner L_M trusts more on the base-learner, which has a more similar resource utilization pattern with the target workload.

6.4.2 Learning from model predictions. When we have more observations of the target task, we can define base-learners' generalizability in terms of how accuracy base-learner can predict the performance (e.g., throughput, internal metrics) of the target task. The challenge here is that performance metrics can differ in scale significantly among various hardware environments in the cloud. Ottertune [6] limits all the mapped workloads under the same hardware in their mapping schema. We solve this problem from an observation: the actual values of the predictions do not matter since we only need to identify the location of the optimum [25]. The fundamental similarity of tasks in an optimization problem means that the tasks have surfaces with similar trends about where the objective is minimized in the configuration space. So if a base-learner can correctly order the configurations according to their performance values, it's considered useful to the meta-learner [13].

Historical base-learner evaluation. Given n_t as the number of target observations, we denote $R_u(l_i)$, as the misranked pairs for metric u i.e. ranking loss for performance metric u that the historical base-learner l_i predicts against the target observation $f_u(\theta, w_{T+1})$:

$$R_u(l_i) = \sum_{j=1}^{n_t} \sum_{k=1}^{n_t} \mathbb{1}\Big(l_i^u(\theta_j) \le l_i^u(\theta_k)\Big) \oplus \Big(f_u(\theta_j, w_{T+1}) \le f_u(\theta_k, w_{T+1})\Big) \quad (9)$$

where $\oplus$ is exclusive-or operator and $u \in \{\text{tps, lat, res}\}$. Consider the objective and constraints, ranking loss for historical base-learner is $\sum_{u \in S}(R_u(l_i))$, which means that if one pair of the configurations is misranked, the ranking loss $R(l_i)$ is increased by one.

Target base-learner evaluation. For target task, the definition above uses in-sample error and can not accurately reflect generalizability of target base-learner l_{T+1}. We estimate generalizability for l_{T+1} with the leave-one-out strategy. We use $l_{T+1,-j}$ donate l_{T+1} with observation $(\theta_j, f(\theta_j, w_{T+1}))$ leaved out. $l_{T+1,-j}$ is constructed by removing the data point from the GP model and kernel hyper-parameters do not need re-estimated. To obtain the ranking loss for target base-leaner, we replace l_i in right side of equal sign in Equation 9 with $l_{T+1,-j}$. Then the loss is the number of misranked pairs that l_{T+1} predicts against real observations in out-of-sample setting. With the dynamic schema, the weight of l_{T+1} is increased when no source tasks can help to prevent "negative transfer" [40].

Finally, each base-learner is weighted with the probability that it is the base-learner with the lowest ranking loss. As $l_i(\theta)$ is a random variable with mean and variance, we can sample from the posterior distribution of $R(l_i)$ and estimate the probability.

6.4.3 Adaptive weight schema. ResTune uses an adaptive weight schema to combine the weights discussed above. The observations of the target task are limited at the beginning. For initialization, it is a common practice to bootstrap optimization by a set of samples. One way is to use Latin Hypercube Sampling [19]. However, we realize that the meta-features generated by workload characterization can give a coarse-grained abstraction about task properties. Therefore, in the initial iterations, we use an ensemble model weighted by meta-features to suggest knobs that are promising according to similar historical tasks. After ResTune collects more observations of the target task, we assign the weights based on the ranking of model predictions, which measure the similarity of tasks in the optimization problem. It provides a dynamic perspective, and the influence of historical base-learners shrinks over iterations. The meta-learner relies more and more on target base-learners gradually.

Table 1: Hardware Configurations for Database Instances

	A	B	C	D	E	F
CPU	48 cores	8 cores	4 cores	16 cores	32 cores	64 cores
RAM	12GB	12GB	8GB	32GB	64GB	128GB

Table 2: Workloads

Name	SYSBENCH	TPC-C	Twitter	Hotel	Sales
Size(G)	10,30,100	13,100	29	14	10
#Thread	64	56	512	256	256
R/W Ratio	7:2	19:10	116:1	19:1	154:1
Request Rate(txn/s)	21K	2K	30K	/	/

7 EVALUATION

In this section, we evaluate the performance of ResTune. Since none of the existing knobs tuning methods optimize resource usages, we carefully modify them to solve the resource-oriented tuning problem defined in this paper. The baselines are presented below.

- Default: The default knobs provided by experienced DBA.
- iTuned: iTuned uses Gaussian Process as a surrogate model and uses the Expected Improvement acquisition function to search for the optimal configuration. We modified iTuned by changing its objective from maximizing the throughput to minimizing the resource utilization, with the algorithm unmodified.
- CDBTune-w-Con: CDBTune utilizes the deep deterministic policy gradient to find the optimal knobs. It relies on a reward function to encourage the recommendation of knobs with minimal latency or maximal throughput. For example, if the latency is larger than the initial value, the reward is negative. If the latency is smaller than the initial value and previous tuning value, the reward is positive. In the other cases, the reward remains zero. To support resource-oriented tuning, we make two modifications to the reward function, namely CDBTune with constraints. First, we encourage the agent to minimize resource usage by replacing latency in the original reward function with resource utilization. Second, we encourage the agent to find feasible knobs. If the reward is positive(decreasing resource usage) but violates SLA, we set the reward zero. If the reward is negative(increasing resource usage) but the SLA is guaranteed, we also set the reward zero.
- OtterTune-w-Con: OtterTune uses a machine-learning pipeline to collect, process, analyze knobs and recommend possible settings by learning from historical data. Ottertune does not consider the SLA requirement, so we replace its acquisition function with CEI in Section 5 we defined in ResTune. We call it OtterTune with constraints. Unlike meta-learning, OtterTune identifies the most similar workload from its repository based on the distance between the internal metrics. It uses the matched data for target workload in a single Gaussian Process (GP) model.
- ResTune-w/o-ML: ResTune without Meta-Learning, which means the data repository is not adopted and learns from scratch.
- ResTune: Our approach that supports solving the resource-oriented problem and uses the meta-learner to boost the tuning.

Setting. We implement ResTune using BoTorch [7] and use version 5.7 of MySQL RDS. The experiments run on cloud servers with six instances, as shown in Table 1. The buffer pool size is fixed when conducting the CPU and I/O resource experiments. We set the buffer

Table 3: Execution Time Breakdown per Iteration Tuning SYSBENCH Workload

Phase	ResTune	ResTune-w/o-ML	iTuned	CDBTune-w-Con	OtterTune-w-Con
Meta-Data Processing	0.653s~1.983s	/	/	/	/
Model Update	0.312s~2.298s	0.649s	0.151s	0.586s	11.347s
Knob Recommandation	5.115s	1.907s	0.912s	0.005s	4.457s
Target Workload Reply	182.237s(95.1%)	182.237s(98.6%)	182.186(99.4%)	182.336s(99.7%)	182.337s(92.0%)
Total Time	191.630s	184.793s	183.245s	182.927s	198.141s

pool size as half of the total memory for all instances. We use 14 knobs to optimize CPU, 6 knobs to optimize memory usage, and 20 knobs to optimize the I/O resource. The knobs are pre-selected as important. We set λ_{tps} and λ_{lat} to the throughput and latency under the DBA's default knobs. For ResTune, We use the static weights learning from meta-feature in the first 10 iterations for initialization , and switch to dynamic weights learning from model predictions afterward. For other BO based methods(ResTune-w/o-ML, iTuned, OtterTune-w-Con), we use Latin Hypercube Sampling(LHS) in the first 10 initial iterations. To avoid the noise of measurement, we accept 5% deviation when evaluating the performance metrics. We run 3 times of each experiment and report the average result.

Workload. Our evaluations are carried out using three benchmarks (SYSBENCH[3] and OLTPBench TPC-C, Twitter [10]) and two real workloads (Hotel Booking and Sales) from our production, as shown in Table 2. The benchmarks both have read and write queries. The data size of the workload is adjusted to a value larger than the buffer pool size on different instances. For SYSBENCH, we use three settings with 150 tables and different table sizes(250K, 1000K, 3000K records). For TPC-C, we use two settings consisting of 200 and 10000 warehouses. Unless otherwise specified, we use SYSBENCH with 250K table sizes and TPC-C with 200 warehouses. We conduct a sensitivity analysis of varying data sizes for SYSBENCH and TPC-C in Section 7.4. For Twitter, we load the data consisting 1500K users. The request rates are counted per second and across all threads and are set for benchmark workloads by observing throughout under DBA's default configuration. For Hotel Booking and Sales, the request rates are not fixed and follow the client request.

Data Repository. We collect workload features and observation histories of 34 past tuning tasks as our meta-data. Workload feature is an embedding vector calculated by workload characterization in Section 6.4.2. The 34 past tuning tasks are from 17 different workloads and 2 hardware environments(instance A and B in Table 1), summing up to 6400 observations. Each observation corresponds to a record of $\left(\theta_i, f_{res}\left(\theta_i\right), f_{tps}\left(\theta_i\right), f_{lat}\left(\theta_i\right)\right)$. We fit 34 historical base-learners for the corresponding tuning tasks. To evaluate the generalizability, we conduct our experiments under three settings: *original setting*, *varying hardware setting*, *varying workloads setting*. Under the *original setting*, we use all the 34 historical base models. We use the existing workload from the data repository as the target workload and not hold out its base model. Under the *varying workloads setting*, we test the adaption ability by holding out the target workload's meta-data and using 32 historical base models of the other workloads. Similarly, we hold out the meta-data in the same hardware environment as the target task under the *varying hardware setting*. We use 17 historical base models(transferring between instance A and B) and 34 historical base models(transferring to the other instances in Table 1) for the tuning task in other instances.

Experiment Outline. First, to evaluate how efficient ResTune finds feasible CPU-optimized knobs, we compare it with the default, ResTune-w/o-ML, OtterTune-w-Con, CDBTune-w-Con, iTuned under the *original setting*. Next, to answer how well ResTune can extract knowledge that accelerates the tuning, we evaluate the generalizability in two aspects. In 7.2.1, we test the ability of ResTune to transfer between different hardware environments. We conduct two groups of experiments under *varying hardware setting*: transferring between instance A and B, and transferring to instances C, D, E and F. In 7.2.2, we compare ResTune with Default, ResTune-w/o-ML, OtterTune-Con under *varying workloads setting*. To further explain why ResTune outperforms other methods, we conduct a case study using the Twitter workload. To illustrate the robustness of ResTune and the correctness of the experiments, we conduct sensitivity analysis on varying request rates and varying data sizes in 7.4. Furthermore, we evaluate ResTune on optimizing the I/O resource and memory resources. We compare ResTune with the default, ResTune-w/o-ML, OtterTune-w-Con, CDBTune-w-Con, iTuned in terms of I/O and the memory usage. Finally, we calculate the 1-year-TCO reduction for different cloud providers using ResTune.

Execution Time. In Table 3, we analyze the detailed execution time of different methods in a single iteration. We calculate execution time into four stages. The Meta-data Processing counts the time processing of the data repository. The Model Update stage calculates the time updating the GP model for BO methods and learning the Actor-Critic network for Reinforcement Learning (RL) methods. For ResTune, the first few iterations use the static weights for initialization that takes 0.312s on average and use the dynamic weights for the following iterations that take 2.298s on average. The Knobs Recommendation stage counts the time for optimizing the acquisition function for BO methods and inference for RL methods. The Target Workload Replay stage stands for the evaluation time in DBMS. We replay the workload for 3 minutes of the benchmark workloads and 5 minutes of the real workloads to capture different types of transactions. The takeaway is that the majority of the time in each iteration of different methods is spent on replaying the target workload. Therefore, it is reasonable to focus on the number of iterations in the following comparison.

7.1 Efficiency Comparison

We evaluate the tuning efficiency of ResTune in various workloads on instance under the *original setting*. Figure 3 shows the CPU utilization of best feasible knobs output by different methods within 200 tuning iterations. The y-ax is current database-wide CPU utilization as percentage, measured per second during the workload replay and calculated on average. We make three observations.

[3]https://github.com/akopytov/sysbench

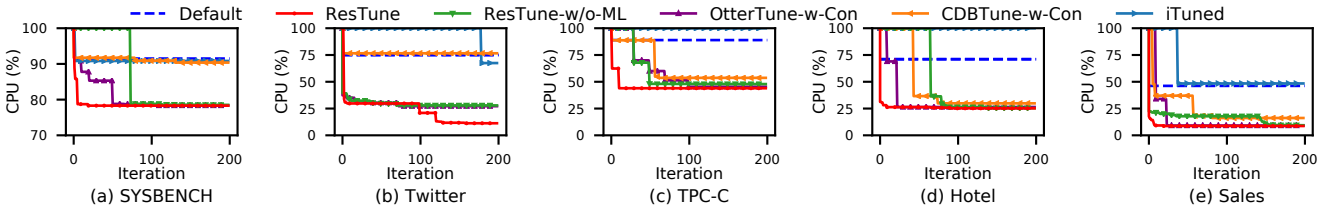

Figure 3: Efficiency Comparison

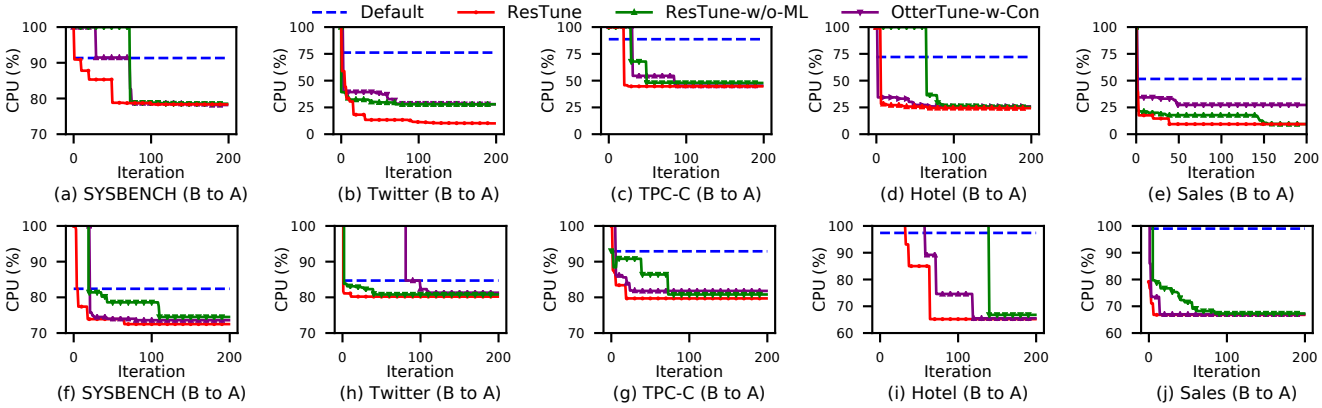

Figure 4: Performance Adapting to Different Hardware Environments

First, ResTune can find CPU-optimized configurations satisfying the SLA requirement. When tuning benchmarks, ResTune can reduce the default CPU usage by 50.1% on average and guarantee the SLA requirement. ResTune achieves more improvement for real workloads, reducing the default CPU utilization by 71.3% and guarantee the SLA requirement. Second, ResTune-w/o-ML performs much better than iTuned and CDBTune-w-Con. All three methods do not utilize meta-data and only learn from the target observations. Compared with iTuned and CDBTune-w-Con, ResTune-w/o-ML achieves better results faster in all workloads. iTuned uses simple Bayesian Optimization without considering constraints. It turns to recommend infeasible configurations with minimum CPU utilization. Such configurations usually restrict throughout and decrease dababase performance. CDBTune-w-Con learns the mapping from internal metrics (state) to configurations (state). In the RL basics, an Markov Decision Process (MDP)[42] formally describes an environment. However, given a target workload, the configuration tuning problem is not necessarily a MDP because the optimal configuration is the same for any internal metrics, leading to the fact that action and state are not necessarily related. Note that for SYSBENCH and Twitter, CBDTune-w-Con can only find the configuration close to the default performance within 200 iterations. Third, with our meta-learning design, ResTune accelerates the tuning with a large margin and makes the tuning time acceptable. On average, ResTune-w/o-ML needs 135 iterations(7.18hours) to find its best results. For Twitter, ResTune finds knobs reducing 59.68% of CPU usage than ResTune-w/o-ML. For the other workloads, ResTune recommends ResTune-w/o-ML's best results within the first 10 iterations and takes 8 iterations(25.6 minutes) to find such configurations on average. Within one hour, ResTune can already find good configurations, while the other methods can not. OtterTune-w-Con is another method that learns from historical data using a mapping strategy, but it falls behind ResTune. ResTune achieves 18.6× speedup than OtterTune-w-Con in SYSBENCH and 7.38× speedup on average. The speedup is contributed to ResTune's meta-learning design, which is explained in Section 7.2.3 in detail.

Table 4: Workload Adaptation on More Instances

Instance			C	D	E	F
SYSBENCH	Improvement	Restune	5.02%	8.13%	17.16%	20.38%
		Restune-w/o-ML	3.34%	7.58%	16.76%	19.96%
	Iteration	Restune	37	64	100	35
		Restune-w/o-ML	57	80	115	53
		Speed Up	35%	20%	14%	34%
TPC-C	Improvement	Restune	4.96%	19.22%	33.26%	47.60%
		Restune-w/o-ML	2.78%	18.28%	33.09%	42.62%
	Iteration	Restune	12	25	45	18
		Restune-w/o-ML	99	47	79	25
		Speed Up	87.87%	46.80%	43.03%	28%

7.2 Evaluation on Generalization

7.2.1 Hardware Adaption. As there are various instances on cloud databases, it's vital whether our proposed methods can adapt to unseen instance types. Learning from different instances is a challenge as the change of hardware setting, such as RAM size, processor capacity, makes the response surfaces between knobs and metrics differ in shape and quantity. We use all the instances in Table 1. First, to compare ResTune and OtterTune-w-Con, we use tuning data collected on instance A to tune the databases on instance B (A to B) and vice versa(B to A). Second, to verify that ResTune can generalize to more instances, we use tuning data collected on instances A and B to train tasks on instances C, D, E and F, respectively. In the second part, we use SYSBENCH(100G) and TPC-C(100G) to ensure that the data size is always larger than the buffer pool size.

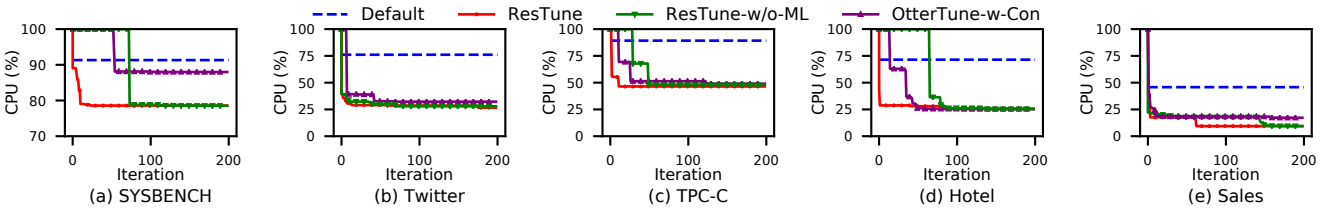

Figure 5: Performance Adapting to Different Workloads

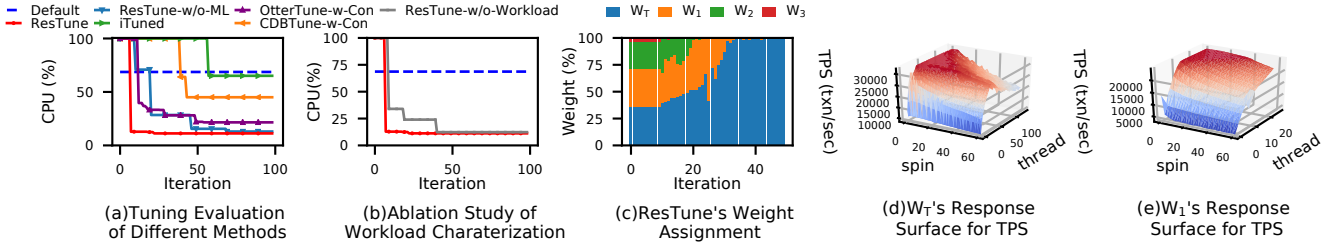

Figure 6: Case Study on Twitter Workload with 3 Tuning Knobs

Figure 4 shows ResTune is better than other methods in all cases. Instance A has more CPU cores and larger memory than instance B, so its SLA requirement is higher. Using meta-data collected on instance B, ResTune successfully finds the feasible configuration faster and better than ResTune-w/o-ML. Nevertheless, for Twitter on both instances and Sales on instance A, OtterTune-w-Con's mapping strategy slows down the tuning process compared with ResTune-w/o-ML and ResTune. Table 4 shows results on ResTune's hardware adaption capabilities to instances C, D, E, and F. We record the improvement(the reduction of best feasible CPU utilization compared to the default) and the iteration (the number of iteration that the best feasible CPU utilization is found). ResTune has better improvement and faster speed than the other baselines. These results show ResTune can reasonably use historical training data to speed up giving an equivalent or superior knobs configuration.

7.2.2 Workload Adaption. We run experiments on 5 target workload on instance A under the *varying workloads setting*. The result is depicted in Figure 5. We can conclude that on the same instance, ResTune outperforms all the other baselines and improves the speed of ResTune-w/o-ML by 3.6× on average.

7.2.3 Analysis. There are three reasons that ResTune outperforms OtterTune-w-Con. First, ResTune identities similar shapes of response surface by ranking loss, even they differ a lot in scale. However, OtterTune-w-Con calculates the average absolute distance of metric vectors between target workload and fails in hardware adaption. Second, ResTune avoids overfitting into historical tasks by using dynamic weight assignment. When the knowledge of historical base-learners is less helpful, ResTune can increase target base-learner's weight up to 100%. However, OtterTune-w-Con keeps matching historical workload and does not have a mechanism to stop when there is no similar workload, which would hinder the optimization (negative transfer). Third, OtterTune-w-Con only reuses single workload data, while ResTune extracts experience through multiple base-learners and ensemble them together.

Table 5: Statistics about Workload Variations

Workload	WT	W1	W2	W3	W4	W5
R/W Ratio	116:1	32:1	19:1	14:1	11:1	9:1
Distance to Wt	0	0.075	0.156	0.191	0.278	0.342
Static Weight	53.57%	46.00%	20.98%	4.80%	0%	0%
Ranking Loss	/	17.93%	22.71%	27.75%	34.04%	60.91%

Table 6: Best Configurations Found by Different Methods

	thread_concurrency	spin_wait_delay	lru_scan_depth	CPU
Default	0	6	1024	75%
Grid Search	17	0	100	14.43%
ResTune	**13**	**0**	**356**	**11.22%**
ResTune-w/o-ML	14	1	100	12.97%
OtterTune-w-Con	30	2	2244	20.59%
CDBTune-w-Con	122	3	180	45.03%
iTuned	43	21	100	65.10%

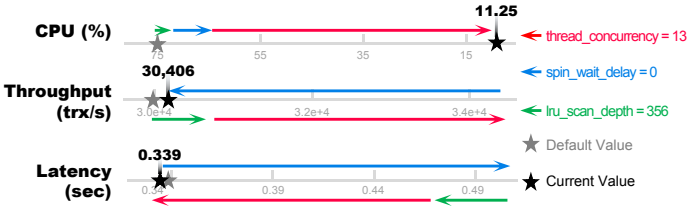

Figure 7: SHAP Path: Features Contributions from Default Knobs

7.3 Case Study

To further explain ResTune's advantages over other methods, we conduct a case study using the Twitter workload. We tune three CPU related knobs, including the maximum number of threads in InnoDB(thread_concurrency, ranging from 100 to 1024), the maximum delay between polls for a spinlock(spin_wait_delay, ranging from 0 to 128), how far down the LRU list a page cleaner thread scans(lru_scan_depth, ranging from 0 to 256). To illustrate the rationale of generalizability, we manually construct a data repository. We create five variations of Twitter by increasing the ratio of INSERT queries gradually, namely $W_1, W_2, ..., W_5$, as shown in Table 5. For each variation, we conduct LHS sampling to collect 200 observations to fit the historical base-learner in the data repository.

We also perform an $8 \times 8 \times 8$ grid search as our known ground-truth. Table 6 shows the knobs recommended by different methods and corresponding CPU utilization. Figure 6(a) shows the tuning process within 100 iterations. We discuss our key findings below.

I. *ResTune optimizes the resource by balancing between database performance and resource utilization.* ResTune finds the best feasible knobs among all the methods. We use SHAP [29], a game-theoretic approach that connects optimal credit allocation with local explanations to analyze the knobs' influence. The SHAP path in Figure 7 explains how each of the recommended knobs helps to get from the default value to the value ResTune outputs (current value). The reduction of CPU usage from 75% to 11.25% is mainly due to setting thread_concurrency and lru_scan_depth to 13 and 0. And the setting of the two knobs help increase the throughput and decrease the latency, while the setting of spin_wait_delay works oppositely. The default value of thread_concurrency is zero, meaning unlimited concurrency. ResTune sets thread_concurrency 13, which has the largest effect for reducing the CPU utilization and improving the performance, as the red arrow shown in the figure. Lower values of thread_concurrency will reduce processors' utility, and higher values will cause performance downgrade due to increased contention on resources, which highly depends on the workload and the hardware environment. ResTune tunes spin_wait_delay to zero, turning off the database's busy polling on spinlocks. Polling on a spinlock will constantly waste CPU time if the lock is held for a long time. The closure of the busy loop saves CPU resources but reduces the database performance. The blue arrow in the figure shows the trade-off between database performance and CPU utilization. Last, ResTune recommends lru_scan_depth to a value contributing to database performance improvement to meet the SLA requirement.

II. *ResTune's workload characterization is effective.* ResTune generates the meta-features of base-learners from workload characterization, and their distances to target workload's meta-feature are used to calculate the weights. Since the historical workloads are made by increasing the ratio of INSERT queries from the target workload, we know the ground truth that W_1 is more similar to the target workload than others. Take the throughput as an example, Figures 6(d) and 6(e) show that W_1 has the similar performance surface with W_T. The static weights in Table 5 also show that the weight of W_1 is the largest. To further dive into the effectiveness of workload characterization, we conduct an ablation study. We replace ResTune's workload characterization pipeline with LHS to generate the first 10 observations, namely ResTune-w/o-Workload. Figure 6(b) shows that compared to ResTune-w/o-Workload, ResTune utilizes the workload characterization to find good results faster.

III. *ResTune generalizes to new tuning tasks by fine-tuning the weights of base-learners.* ResTune detects similar base-learners by ranking loss and assigns them larger weights. Consequently, the optimal regions of these base-learners are exploited with high probability. Table 5 presents the mean of ranking loss for the historical base-learners, which is calculated as how many ranking pairs are incorrectly ranked in the percentage of total pairs. Among the historical base-learners, W_1 is most similar to the target, followed by W_2 to W_5. For the target base-learner, its ranking loss decreases gradually as it collects more observations of the target task. Figure 6(c) shows the weight assignment of ResTune in the first 50 iterations. Within 25 iterations, similar historical base-learners have larger weights. So base-learner W_1 indicates the optimal region, accelerating the tuning process. With more target observations, the target base-learner's weight dominates, and overfitting is avoided. ResTune outperforms other baselines and even the grid search, as the search stepsize of grid search can be smaller.

7.4 Sensitivity Analysis

We illustrate the robustness of ResTune by varying the request rate and varying the data size.

7.4.1 Varying Request Rate. We conduct eight experiments on optimizing CPU usage for SYSBENCH and TPC-C, respectively. For SYSBENCH, we vary the request rate from 16000 to 23000. For TPC-C, we vary the request rate from 1500 to 2200. Figure 8 shows that ResTune could accomplish similar improvement regardless of the request rate setting compared with the default configuration. Surprisingly, we find that the knobs ResTune outputs under a certain request rate can be transferred to the setting with different request rates, achieving similar improvement with the red line in Figure 8. Note that if the request rate decreases further, the room for optimization decreases since the CPU utilization is too low.

7.4.2 Varying data size. As Table 7 shows, we carry out 5 experiments on optimizing CPU utilization for TPC-C workload with different data sizes, by setting the number of warehouses. In each experiment, CPU utilization drops significantly after tuning knobs. It is worth noting that when the amount of data is small, the decline in CPU utilization becomes smaller. This is because CPU resource is limited in this scenario, and resource optimization problems cannot be simply solved by tuning configuration knobs. It is recommended to consider further increasing CPU resources. On the other hand, when the amount of data is large, the decline in CPU utilization will also decrease because of the lower hit ratio. Default CPU utilization is also reduced accordingly. In this scenario, consider optimizing memory-related knobs or increasing memory resources.

7.5 Tuning other types of Resources

In previous sections, we mainly focus on evaluating ResTune on optimizing the CPU usage. The reason is that we found in the majority of OLTP workloads in production, memory, and I/O are usually not the bottleneck since the hot data are mostly cached. But ResTune can be extended to optimize more resources. In this section, we evaluate the performance of ResTune on optimizing the I/O and the memory resources. We conduct experiments on instance E, tuning SYSBENCH and TPC-C under the *varying workloads setting*. We use the tuning observations on SYSBECNH as meta-data in the data repository to tune TPC-C(SYSBENCH to TPC-C) and vice versa(TPC-C to SYSBENCH). To mimic the I/O heavy scenario, we fix the buffer pool size to 16G and initialize the data with 100G for TPC-C (10000 warehouses), 30G for SYSBENCH (150 tables, 100w records in each table). The hit ratio under the default configuration is 93.2% for TPC-C and 97.5% for SYSBENCH. We use the same workload setting for the memory experiment since it is more challenging to reduce the memory resource in I/O heavy cases. The only difference for memory experiments is that we do not fix the buffer pool size and use it as a tuning knob. The comparison methods are the same with CPU experiments, with

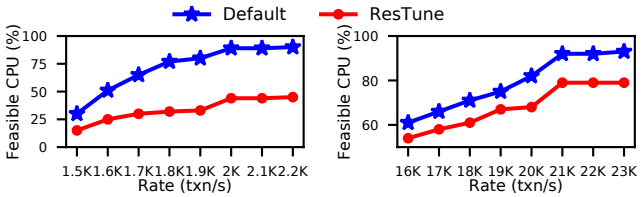

Figure 8: Sensitivity Analysis of Request Rate

Table 7: Sensitivity Analysis of Data Size

#Warehouses	Size(GB)	Hit Ratio	Default CPU	best CPU	Improvement
100	7.29	0.996	90.21	58.51	35.13%
200	16.26	0.995	87.78	43.95	49.93%
500	35.26	0.991	88.60	40.48	50.77%
800	56.59	0.984	78.59	34.53	58.52%
1000	117.06	0.946	46.00	36.62	44.80%

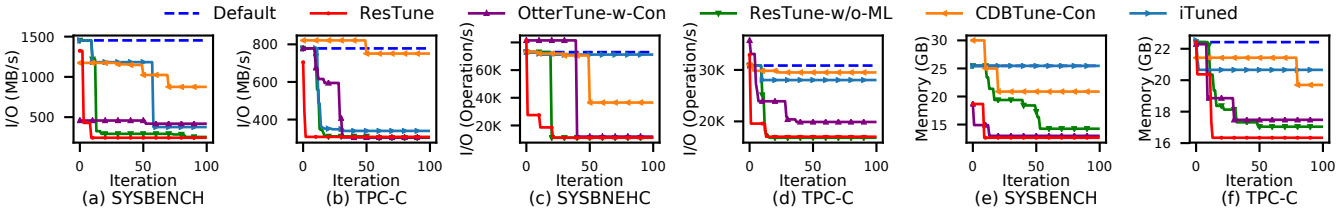

Figure 9: Tuning Other Types of Resources

changing the objective from minimizing the CPU utilization to the I/O and memory usage.

7.5.1 I/O. There are two important metrics that measure the I/O utilization with different aspects. BPS(Bytes Per second) measures the total bytes of read and write operations in each second. IOPS(I/O operations per second) measures the number of invocations of read and write operations in each second. We evaluate ResTune by setting BPS and IOPS as objective, respectively, and tune 20 selected knobs. The first four figures in Figure 9 show the results. We can see that ResTune reduces 60% - 80% of BPS under the default configuration and reduce 84% - 90% of IOPS under the default configuration and outperforms others.

7.5.2 Memory. We conduct two experiments by tuning 6 memory-related knobs. Figure 9 shows that for TPC-C, ResTune reduces the total memory usage of DBMS from 22.5G to 16.34G. For SYSBENCH, ResTune reduces the total memory usage of DBMS from 25.4G to 12.64G, outperforming other baselines. Compared with ResTune-w/o-ML, ResTune finds good configurations with 15 iterations, showing the advantage of meta-learning.

7.6 TCO Analysis

To illustrate the benefit from cloud providers' perspective, we estimate the reduction of Total Cost of Ownership (TCO) using ResTune. We focus on the RDS MySQL product and calculate corresponding TCO reduction among AWS, Azure, and Aliyun using their online calculators [3–5]. It is not easy to find the exact same instance type of our experiments among all three cloud environments. Therefore, we estimate the TCO reduction for each CPU and every GB of the memory instead. For example, the 1-year-TCO for the RDS MySQL in Aliyun of with 8 Core 16GB and 500G SSD storage is $4032 and with 4 Core 16GB is $3852. So the TCO reduction per Core in Aliyun is ($4032 − $3852)/4 = $45. We calculate the 1-year-TCO reduction according to our previous experimental results. Table 8 shows the detailed reduction of 1-year-TCO of two workloads from 6 different instance types. The TCO reduction is averaged of the reductions among AWS, Azure, and Aliyun. Table 9 shows the TCO reduction

Table 8: 1-year-TCO Reduction Optimizing CPU Usage

Workload		InstanceA	InstanceB	InstanceC	InstanceD	InstanceE	InstanceF
SYSBENCH	Original CPU	43 Cores	7 Cores	4 Cores	16 Cores	29 Cores	58 Cores
	Optimized CPU	21 Cores	6 Cores	4 Cores	15 Cores	24 Cores	46 Cores
	Avg TCO↓	$8,749	$398	$0	$398	$1,988	$4,772
TPCC	Original CPU	44 Cores	8 Cores	4 Cores	16 Cores	30 Cores	52 Cores
	Optimized CPU	38 Cores	7 Cores	4 Cores	13 Cores	20 Cores	27 Cores
	Avg TCO↓	$2,386	$398	$0	$1,193	$3,977	$9,942

Table 9: 1-year-TCO Reduction Optimizing Memory on Instance E

	Original MEM	Optimized MEM	TCO↓(AWS)	TCO↓(Azure)	TCO↓(Aliyun)
SYSBENCH	25.4GB	12.64GB	$983	$855	$2144
TPCC	22.5GB	16.34GB	$475	$412	$1035

optimizing memory on instance E. Note that the originally used resource might be less than the total resource of the instance, and the reduced TCO is calculated based on the originally used resource. We omit the TCO reduction of the I/O resource since not all online calculators support customizing the IOPS/BPS.

8 CONCLUSION

We present ResTune to optimize resource utilization for cloud databases automatically. By leveraging constrained Bayesian optimization, ResTune can meet the SLA requirement and saving CPU, memory, and I/O resources. To make tuning time acceptable for end-users, ResTune implements a meta-learning approach to extract experience from historical tuning tasks. ResTune can also be applied across heterogeneous hardware environments. The evaluations on benchmarks and real workloads show that ResTune can reduce 65% of CPU, 87% of I/O and 39% of memory on average from the DBA configuration. Comparing to the state-of-the-art systems, ResTune can find better configurations and achieves up to ~ 18× speedups finding the configuration with the same resource utilization.

ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China (NSFC)(No. 61832001), Alibaba Group through Alibaba Innovative Research Program and National Key Research, and the Beijing Academy of Artificial Intelligence.

REFERENCES

[1] 2018. Amazon RDS Under the Hood. https://aws.amazon.com/blogs/database/amazon-rds-under-the-hood-single-az-instance-recovery/.

[2] 2019. High AWS EC2 CPU utilization should be avoided. https://www.cloudmanagementinsider.com/avoid-high-aws-ec2-cpu-utilization/.

[3] 2020. Alibaba Cloud Computing. https://www.alibabacloud.com/pricing-calculator.

[4] 2020. Amazon Web Services (AWS). https://calculator.aws/#/.

[5] 2020. Microsoft Azure. https://azure.microsoft.com/en-us/pricing/tco/calculator/.

[6] 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.

[7] Maximilian Balandat, Brian Karrer, Daniel R. Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, and Eytan Bakshy. 2020. BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization. In *NeurIPS*.

[8] Rémi Bardenet, Mátyás Brendel, Balázs Kégl, and Michèle Sebag. 2013. Collaborative hyperparameter tuning. In *ICML (2) (JMLR Workshop and Conference Proceedings, Vol. 28)*. JMLR.org, 199–207.

[9] Sudipto Das, Vivek R. Narasayya, Feng Li, and Manoj Syamala. 2013. CPU sharing techniques for performance isolation in multi-tenant relational database-as-a-service. *Proceedings of the VLDB Endowment* 7, 1 (2013), 37–48.

[10] 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.

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

[12] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and Robust Automated Machine Learning. In *NIPS*. 2962–2970.

[13] Matthias Feurer, Benjamin Letham, and Eytan Bakshy. 2018. Scalable meta-learning for bayesian optimization using ranking-weighted gaussian process ensembles. In *AutoML Workshop at ICML*, Vol. 7.

[14] Jacob R. Gardner, Matt J. Kusner, Zhixiang Eddie Xu, Kilian Q. Weinberger, and John P. Cunningham. 2014. Bayesian Optimization with Inequality Constraints. In *ICML (JMLR Workshop and Conference Proceedings, Vol. 32)*. JMLR.org, 937–945.

[15] Michael A. Gelbart, Jasper Snoek, and Ryan P. Adams. 2014. Bayesian Optimization with Unknown Constraints. In *UAI*. AUAI Press, 250–259.

[16] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. 2017. Google Vizier: A Service for Black-Box Optimization. In *KDD*. ACM, 1487–1495.

[17] Djoerd Hiemstra. 2000. A probabilistic justification for using tf x idf term weighting in information retrieval. *Int. J. Digit. Libr.* 3, 2 (2000), 131–139.

[18] 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.

[19] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (Eds.). 2019. *Automatic Machine Learning: Methods, Systems, Challenges*. Springer.

[20] Dean Jacobs and Stefan Aulbach. 2007. Ruminations on Multi-Tenant Databases. In *BTW (LNI, Vol. P-103)*. GI, 514–521.

[21] Yoon Kim, Yacine Jernite, David A. Sontag, and Alexander M. Rush. 2016. Character-Aware Neural Language Models. In *AAAI*. AAAI Press, 2741–2749.

[22] Andreas Krause and Cheng Soon Ong. 2011. Contextual Gaussian Process Bandit Optimization. In *NIPS*. 2447–2455.

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

[24] Herbert K. H. Lee, Robert B. Gramacy, Crystal Linkletter, and Genetha A. Gray. 2011. Optimization Subject to Hidden Constraints via Statistical Emulation. *Pacific Journal of Optimization* 7, 3 (2011).

[25] Christiane Lemke, Marcin Budka, and Bogdan Gabrys. 2015. Metalearning: a survey of trends and technologies. *Artif. Intell. Rev.* 44, 1 (2015), 117–130.

[26] Bofang Li, Aleksandr Drozd, Yuhe Guo, Tao Liu, Satoshi Matsuoka, and Xiaoyong Du. 2019. Scaling Word2Vec on Big Corpus. *Data Sci. Eng.* 4, 2 (2019), 157–175.

[27] 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.

[28] 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)*.

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

[30] E. A. Nadaraya. 1964. On Estimating Regression. *Theory of Probability and Its Applications* 9 (1964), 141–142.

[31] Carl Edward Rasmussen and Christopher K. I. Williams. 2006. *Gaussian processes for machine learning*. MIT Press.

[32] Nicolas Schilling, Martin Wistuba, Lucas Drumond, and Lars Schmidt-Thieme. 2015. Hyperparameter Optimization with Factorized Multilayer Perceptrons. In *ECML/PKDD (2) (Lecture Notes in Computer Science, Vol. 9285)*. Springer, 87–103.

[33] Nicolas Schilling, Martin Wistuba, and Lars Schmidt-Thieme. 2016. Scalable Hyperparameter Optimization with Products of Gaussian Process Experts. In *ECML/PKDD (1) (Lecture Notes in Computer Science, Vol. 9851)*. Springer, 33–48.

[34] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. 2016. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proc. IEEE* 104, 1 (2016), 148–175.

[35] Muhammad Bilal Sheikh, Umar Farooq Minhas, Omar Zia Khan, Ashraf Aboulnaga, Pascal Poupart, and David J. Taylor. 2011. A bayesian approach to online performance modeling for database appliances using gaussian models. In *ICAC*. ACM, 121–130.

[36] Jasper Snoek. 2014. Bayesian Optimization and Semiparametric Models with Applications to Assistive Technology.

[37] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *NIPS*. 2960–2968.

[38] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Jose Andrade. 2018. P-Store: An Elastic Database System with Predictive Provisioning. In *SIGMOD Conference*. ACM, 205–219.

[39] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. 2019. iBTune: Individualized Buffer Tuning for Large-scale Cloud Databases. *Proc. VLDB Endow.* 12, 10 (2019), 1221–1234.

[40] Matthew E. Taylor and Peter Stone. 2009. Transfer Learning for Reinforcement Learning Domains: A Survey. *J. Mach. Learn. Res.* 10 (2009), 1633–1685.

[41] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In *KDD*. ACM, 847–855.

[42] Martijn van Otterlo and Marco A. Wiering. 2012. Reinforcement Learning and Markov Decision Processes. In *Reinforcement Learning*. Adaptation, Learning, and Optimization, Vol. 12. Springer, 3–42.

[43] Joaquin Vanschoren. 2018. Meta-Learning: A Survey. *CoRR* abs/1810.03548 (2018).

[44] Ricardo Vilalta and Youssef Drissi. 2002. A Perspective View and Survey of Meta-Learning. *Artif. Intell. Rev.* 18, 2 (2002), 77–95.

[45] Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. 2015. Sequential Model-Free Hyperparameter Tuning. In *ICDM*. IEEE Computer Society, 1033–1038.

[46] Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. 2016. Two-Stage Transfer Surrogate Model for Automatic Hyperparameter Optimization. In *ECML/PKDD (1) (Lecture Notes in Computer Science, Vol. 9851)*. Springer, 199–214.

[47] Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. 2018. Scalable Gaussian process-based transfer surrogates for hyperparameter optimization. *Mach. Learn.* 107, 1 (2018), 43–78.

[48] Dani Yogatama and Gideon Mann. 2014. Efficient Transfer Learning Method for Automatic Hyperparameter Tuning. In *AISTATS (JMLR Workshop and Conference Proceedings, Vol. 33)*. JMLR.org, 1077–1085.

[49] 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.

[50] Wentao Zhang, Jiawei Jiang, Yingxia Shao, and Bin Cui. 2020. Snapshot boosting: a fast ensemble framework for deep neural networks. *Sci. China Inf. Sci.* 63, 1 (2020), 112102.

[51] 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.

[52] Zainab Zolaktaf, Mostafa Milani, and Rachel Pottinger. 2020. Facilitating SQL Query Composition and Analysis. In *SIGMOD Conference*. ACM, 209–224.