

A Unified and Efficient Coordinating Framework for Autonomous DBMS Tuning

Xinyi Zhang^{*†‡}
Peking University &
Alibaba Group
zhang_xinyi@pku.edu.cn

Zhuo Chang^{*†‡}
Peking University &
Alibaba Group
z.chang@pku.edu.cn

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

Yang Li[†]
Peking University
liyang.cs@pku.edu.cn

Jia Chen[†]
Peking University
MaCasK@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

Recently using machine learning (ML) based techniques to optimize the performance of modern database management systems (DBMSs) has attracted intensive interest from both industry and academia. With an objective to tune a specific component of a DBMS (e.g., index selection, knobs tuning), the ML-based tuning agents have shown to be able to find better configurations than experienced database administrators (DBAs). However, one critical yet challenging question remains unexplored – how to make those ML-based tuning agents work collaboratively. Existing methods do not consider the dependencies among the multiple agents, and the model used by each agent only studies the effect of changing the configurations in a single component. To tune different components for DBMS, a coordinating mechanism is needed to make the multiple agents be cognizant of each other. Also, we need to decide how to allocate the limited tuning budget (e.g., time and resources) among the agents to maximize the performance. Such a decision is difficult to make since the distribution of the reward (i.e., performance improvement) corresponding to each agent is unknown and non-stationary. In this paper, we study the above question and present a unified coordinating framework to efficiently utilize existing ML-based agents. First, we propose a message propagation protocol that specifies the collaboration behaviors for agents and encapsulates the global tuning messages in each agent’s model. Second, we combine Thompson Sampling, a well-studied reinforcement learning algorithm with a memory buffer so that our framework can allocate the tuning budget judiciously in a non-stationary environment. Our framework defines the interfaces adapted to a broad class of ML-based tuning

agents, yet simple enough for integration with existing implementations and future extensions. Based on extensive evaluations, we show that this framework can effectively utilize different ML-based agents and find better configurations with 1.4–14.1× speedups on the workload execution time compared with baselines.

KEYWORDS

performance tuning, ML for data management

ACM Reference Format:

Xinyi Zhang, Zhuo Chang, Hong Wu, Yang Li, Jia Chen, Jian Tan, Feifei Li, and Bin Cui. 2022. A Unified and Efficient Coordinating Framework for Autonomous DBMS Tuning. In *Proceedings of SIGMOD ’23: ACM SIGMOD/PODS Conference (SIGMOD ’23)*. ACM, New York, NY, USA, 15 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 INTRODUCTION

Database management systems (DBMSs) are playing critical roles in a broad spectrum of data-intensive applications. The performance of a DBMS depends on multiple configurable components, e.g., configuration knobs, index settings, and materialized views. Tuning the DBMS is crucial to obtain high performance, but the process is not trivial. As a common practice, database administrators (DBAs) put considerable effort into finding appropriate settings for the DBMS. However, since the NP-hard nature [41] of the tuning problem, manual manipulation can not handle such complex tuning tasks easily. Due to the shift to cloud environments, there are large-scale instance deployments with increasingly diverse workloads in cloud databases. The manual tuning fails to scale well.

Fortunately, machine learning (ML) techniques could help to overcome this challenge since they can model complex functions and automate the performance tuning process. Researchers have been extensively working on approaches that use ML techniques to optimize database systems (i.e., ML-based tuning agents). Figure 1a presents the number of published research focusing on the ML-based tuning agents for DBMS in recent five years. These works include index selection [15, 22, 23, 35, 38, 47, 52, 54–56, 63, 63, 64, 73], knobs tuning [6, 10, 11, 20, 24, 31–33, 42, 57, 68, 70, 71], view generation [27, 46, 65, 67], query optimization [13, 28, 36, 36, 50, 60, 66, 72], and data partition [17–19, 26, 29, 74]. Given a workload, the ML-based tuning agents aim to improve the database’s performance based on certain objective functions (e.g., higher throughput and lower latency). For example, to tune the configuration knobs in

^{*}Xinyi Zhang and Zhuo Chang contribute equally to this paper

[†]School of CS & Key Laboratory of High Confidence Software Technologies, Peking University

[‡]Database and Storage Laboratory, Damo Academy, Alibaba Group

[§]Institute of Computational Social Science, Peking University(Qingdao), China

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](https://permissions.acm.org).
SIGMOD ’23, June 18–23, 2023, Seattle, WA

© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...\$15.00
<https://doi.org/XXXXXXX.XXXXXXX>

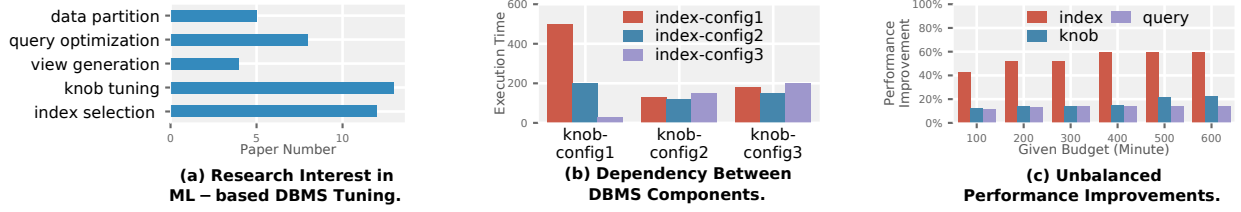


Figure 1: Motivating Examples. Figure 1a shows the number of published papers on ML-based tuning agents from 2017 to 2022. Figure 1b presents the query execution times under different configuration settings for knob and index components. Figure 1c presents the performance improvement of three tuning agents given different tuning budgets on TPC-H workload.

a DBMS, a knobs tuning agent utilizes a ML model to suggest a promising configuration and applies it to the DBMS. Then the agent updates its model based on the performance of the configuration to learn the underlying objective function and to improve its tuning policy.

Although the ML-based tuning agents have demonstrated superior performance and efficiency compared to manual tuning [30, 39], they always focus on the tuning for a single configurable component and neglect the dependencies between agents. However, the performance of a DBMS is influenced by multiple components. Optimal configuration choices for one component would depend on the configurations of the other [51]. For example, it is better to set a larger query cache, smaller buffer size when no indexes are built, and smaller query cache and larger buffer pool when suitable indexes are built [51]. Thus, running a standalone agent is very likely to recommend sub-optimal configurations. Figure 1b shows a concrete example when running Q19 from TPC-H workload. We observe that the optimal configuration is the combination of knob configuration 1 and index configuration 3. However, when knob configuration 2 or 3 is applied to DBMS, the index agent would suggest sub-optimal configurations (i.e., index configuration 2). And when running a tuning agent, the configurations in other components usually are not the ideal ones (e.g., knob configuration 1 in the example). Hence, tuning multiple components together would own more advantages compared with the standalone tuning.

The multi-component tuning is essentially an optimization problem that navigates a joint configuration space composed of the subspace for individual components. The composite configuration space is huge and complex, which leads to scalability issues related to the curse of dimensionality [45]. Besides, since the configuration spaces for different components are distinct (e.g., binary for index selection, continuous for knob tuning, and tree-structured for query rewrite), it is impossible to jointly optimize them directly. Meanwhile, space decomposition has shown promising performance when solving a high-dimensional optimization problem [43–45, 48]. Given the above facts, we decompose the joint space according to the corresponding components and resort to the solvers of existing tuning agents to tune the corresponding components. Consequently, we can take full advantage of the latest research on ML-based tuning agents customized for a specific component. To this end, we seek to answer a question — “how to coordinate existing ML-based tuning agents to configure multiple components in a DBMS?”

There are three main challenges behind this question. First, there are emerging ML-based tuning agents with advanced features for

different DBMS components. From a system perspective, without a high-level abstraction of the agents and a general solution, we can not unitize them for multi-component tuning. *Can we design a unified solution to conveniently support existing ML-based tuning agents and the integration with future extensions (C1)?* Second, given the dependencies among different components, the tuning agents should be able to adjust their tuning policies according to others’ tuning decisions. However, existing studies tune the DBMS in a standalone way – the ML models of tuning agents operate under the assumption that the configurations of other components are fixed. The agents could not make accurate predictions, if other agents modify their controlled configurations [51], thus failing to find global promising configurations. *To leverage existing agents to tune multiple components, we need a mechanism to make the agents communicate and coordinate with each other (C2).* Third, running the ML-based agent is expensive, as it needs time and resources to observe the effects of configurations. However, the distribution of reward (i.e., performance improvement) corresponding to each agent is unknown. And, under a given tuning budget (e.g., time), the agents’ reward is unbalanced and non-stationary (decaying in most cases, i.e., diminishing marginal utility), as shown in Figures 1c. *Given these facts, we need to explore a judicious policy that can efficiently allocate budget to the promising tuning agents (C3).*

To address the above challenges, we propose a unified and efficient coordinating framework, UniTune, for orchestrating the ML-based tuning agents. We first analyze existing agents for DBMSs and classify them into three categories: Bayesian Optimization (BO) based, Reinforcement Learning (RL) based, and RL-estimator based approaches. We abstract the logic of different tuning agents and define interfaces to support a wide variety of existing agents and for future extension (addressing C1). Second, to break the standalone tuning limitation, we propose a message propagation protocol that enables collaboration among agents. It specifies how one agent broadcasts its tuning decision as a message and how other agents receive and process the message. To update the tuning policy according to the tuning messages, we encapsulate the messages in each agent’s model in the form of context features (addressing C2). Third, we formulate the budget allocation problem as a multi-arm bandit problem with each agent referred to an arm and propose a Thompson Sampling based strategy to select promising agents. This strategy maximizes the tuning performance based on historic observations by judiciously balancing the trade-off between exploitation and exploration. To avoid the selection misguided by the changing reward in non-stationary environments, we propose a

strategy that utilizes a memory buffer to discard the out-of-date observations (addressing C3).

To the best of our knowledge, UniTune is the first unified framework using ML techniques to tune multiple components in a DBMS. In summary, we make the following contributions.

- To tune multiple components in a DBMS efficiently, we discuss how to utilize the existing ML-based tuning agents and design a unified coordinating framework with high-level abstraction and flexible interfaces.
- To enable collaboration among agents, we propose a message propagation protocol that specifies their message sharing behaviors and encapsulates tuning messages in agents' models.
- To allocate the limited tuning budget among agents, we propose a Thompson Sampling based strategy to select promising agents based on historical observations. It tackles the non-stationary nature of reward via a memory buffer design.
- We conduct extensive evaluations on synthetic and real workloads. The result shows that UniTune can support different ML-based tuning agents and recommend better configurations with 1.4~14.1 \times speedups on the workload execution time compared with baselines.

The remainder of the paper is organized as follows. We discuss related work in Section 2. We formulate the multi-component tuning problem and review existing ML-based tuning agents in Section 3. Then we provide a high-level abstraction for the ML-based tuning agents and give an overview in Section 4. In Section 5, we introduce a message propagation protocol to enable the collaboration of tuning agents. In Section 6, we provide a policy to allocate the limited tuning budget among agents. Finally, we report the evaluation results in Section 7 and end this paper with a conclusion.

2 RELATED WORK

Recently, configuring DBMSs automatically has attracted intensive interest in both industry and academia. They aim to automate the database tuning tasks and search for better configurations in a data-driven way [58]. They adopt Reinforcement Learning (RL) and Bayesian Optimization (BO) algorithms to tune a DBMS in a trial-and-error manner. Prior works typically focus on a specific tuning task such as index selection, knobs tuning, query rewrite, view generation, and data partition.

Index Selection. Indexes on appropriate columns are vital to speed up query execution [49]. ML techniques are utilized to select proper indexes from a large number of possible index combinations. For example, SmartIX [47], MANTIS [56] and AutoIndex [73] adopt RL, and DBA-bandits [52] adopts C^2UCB algorithm in BO framework.

Knobs Tuning. A DBMS has hundreds of configurable knobs, affecting its performance [69]. To replace the manual tuning that depends on DBA's experience, the DB community has developed lots of ML-based methods to automate this process, such as BO-based [6, 11, 16, 20, 33, 37, 70, 71] and RL-based [10, 42, 68] methods.

Query Rewrite aims to transform a query into an equivalent one but with higher performance. It is an NP-hard problem [21] with numerous rewrite orders (e.g., different operators and rules). LearnedRewrite [72] uses a regression model to estimate the benefit of a rewrite node. And a light-weight RL-based agent, Monte Carlo Tree

Search (MCTS) is adopted to interact with the regression model, searching for a better-rewritten query.

View Generation. Materialized views could save redundant computations among queries that share equivalent sub-queries, based on the space-for-time trade-off principle. Two approaches [27, 67] use a regression model to estimate the benefit of the different view candidates and queries and suggest view-query pairs for a given workload via a RL-based agent with deep Q-learning algorithm.

Data Partition. Partitioning a database can greatly improve the performance of analytical workloads since data-intensive queries can be assigned to multiple machines. Many approaches [17–19, 29] utilize RL agents to explore different partition keys. And Li et al. [29] implements a regression model to estimate partition benefits.

We focus on designing a unified coordinating framework of the ML-based tuning agents to tune multiple DBMS components. The closest work to us is one recent research, UDO [59]. It proposes to use RL agents to optimize more DBMS components. It separates the configurations as heavy and light – heavy parameters have high reconfiguration overheads (e.g., indexes) and light parameters have negligible overheads (e.g., knobs). It uses a two-layer loop to reduce reconfiguration overheads. In the outer layer, a RL agent suggests and applies a heavy parameter. Then, in the inner layer, another RL agent is initialized and iterates for a number of iterations to find a suitable setting for the light parameters. However, we should allocate the tuning budget according to the expected utility of tuning agents, instead of the reconfiguration overheads. The two-layer schedule causes a fixed budget allocation pattern: less tuning budget spent on the outer agent. The fixed schedule leads to a bad overall performance, as shown in our evaluation in Section 7.2. Different from our general framework, UDO is not designed to coordinate existing tuning agents. It focuses on customized algorithms to tune the DBMSs. For example, it proposes an MCTS variant, delayed-HOO to suggest heavy parameters, and a planner to reduce re-configuration overheads by carefully arranging evaluation orders of index configurations. Those customized algorithms are orthogonal to our work, and they can be considered as another tuning agent and integrated into our framework.

3 PRELIMINARY

In this section, we formulate the problem and review the existing ML-based tuning agents for DBMS.

3.1 Terminology and Problem Statement

We first introduce the associated terminology and then define the multi-component tuning problem.

Tuning Agent. An agent is an external system focusing on tuning a specific component of DBMS to improve pre-defined metrics.

Example 1. Modern DBMSs have several configurable components that affect their runtime execution and performance, including (1) physical design (e.g., index, view, and data partitioning), (2) configuration knobs, (3) query design (e.g., rewriting a query externally). An agent is usually responsible for configuring one of these components. For example, to minimize the execution time of a given workload, a knobs-tuning agent will search for the suitable

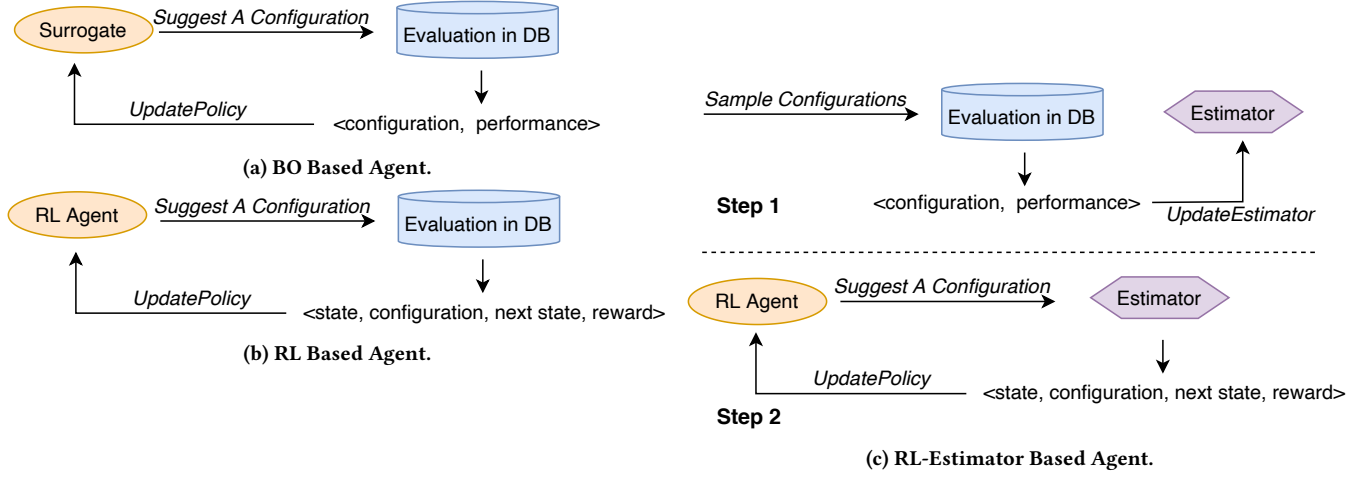


Figure 2: Three Categories of ML-Based Tuning Agents.

knobs' values and a view-generation agent will select proper materialized views for queries in the workload. We denote the agent for component i as A_i .

Configuration Subspace. A configuration subspace Θ_i is the set of all possible configurations for a component i . And we denote the configuration value in Θ_i as θ_i . And the domain of a configuration subspace depends on the nature of the corresponding component and the formulation made by the agent.

Example 2. For knob tuning, its domain could be a mix of continuous variables (e.g., *innodb_buffer_pool_size*), and categorical variables (e.g., *innodb_stats_method*). For index selection, its domain could be a set of binary values representing whether the indexes are created or not. And the domain can even be a set of tuples. For example, the query rewrite agent makes an atomic tuning decision (o, r) , representing applying rewrite rule r on operator o .

Configuration Space. A configuration space Θ is a composition of the configuration subspace of m configurable components, i.e., $\Theta = \Theta_1 \times \Theta_2 \times \dots \times \Theta_m$.

Tuning Budget. The tuning budget is the total amount of cost that is allowed to spend on tuning a DBMS. We can measure the cost from different perspectives, such as time, resource, or money which are positively correlated. For simplicity, we measure the cost in terms of time in this paper. We use sub-budget as a hyper-parameter to control the unit of budget spent for executing an agent.

Problem Statement. Considering m DBMS components, the goal of multi-component tuning is to optimize a user predefined metric by running ML-based tuning agents under a limited tuning budget:

$$\arg \max_{\theta_1, \dots, \theta_m} f(\theta_1, \dots, \theta_m; W),$$

where $\theta_1, \dots, \theta_m \in \Theta_1 \times \Theta_2 \times \dots \times \Theta_m$ is the configuration value suggested by the corresponding agents. W corresponds to the given workload. And f is the database performance metrics, which can be any pre-defined objective, e.g., throughput and 99%th percentile latency. Note that the result of $f(\cdot)$ can be observed after the evaluations in the DBMS (e.g., stress testing).

3.2 Existing ML-based Agents

We categorize existing ML-based agents into three types: Bayesian Optimization (BO) based, Reinforcement Learning (RL) based, and RL-estimator based.

BO based agents formulate the tuning task as a black-box optimization problem and update its tuning policy by interacting with the database system, as shown in Figure 2a. To solve the black-box optimization, the agent works iteratively: (1) suggest the next configuration to evaluate by computing an acquisition function value that measures the utility of candidate points, (2) evaluating the suggested configuration by interacting with the DBMS, and (3) updating a probabilistic surrogate model that describes the relationship between configurations and their performance.

RL based agents formulate the tuning task as a Markov Decision Process (MDP) and update its tuning policy by interacting with the database system, as shown in Figure 2b. An MDP models a problem as a multi-step process. At each step, the agent observes the current state s and chooses an action $a \in A(s)$ based on a policy. The action results in a reward $r(s, a)$ and a change of state to s' . The RL based agents aim to find a decision-making policy that maximizes cumulative observed reward.

RL-estimator based agents also formulate the tuning task as an MDP, but the RL agents interact with a pre-trained estimator to update its tuning policy. The estimator could estimate the utility of a given action. Existing work adopts a two-step manner to train estimators and RL agents, as shown in Figure 2c. In step 1, the estimator is trained in an offline manner. And the training data is obtained by randomly sampling configurations and observing their performance by interacting with the DBMS. In step 2, the RL agent updates the policy by interacting with the estimator instead of evaluating in DBMS.

Scope Illustration. We focus on configuring multiple components in a DBMS via a unified framework for different ML-based tuning agents. To make our framework clear and general, we clarify some necessary constraints. (1) We limit the scope of our framework to the support of the *external* agents, which configure the database

components exposed by DBMS’s APIs without having to modify the DBMS’s internal implementation. The external approach is easy to apply to existing DBMSs without software engineering effort to retrofit their architecture. (2) At the algorithm level, we focus on offline tuning. Online tuning can be implemented via a clone and parallelization scheme to stress-test workloads on multiple cloned instances and apply the safe one in the online database [10]. (3) Currently, we let users choose one algorithm for tuning one component (we provide default options for each component). For future work, we can support automatic algorithm selection for each component, for instance, deciding which algorithm to use for the knobs component from the candidates such as OtterTune [6], CDBTune [68] and CGPTuner [11].

4 OVERVIEW

In this section, we highlight the core abstractions in UniTune and discuss the user interface and workflow.

4.1 Core Abstractions

Our framework provides a high-level abstraction for the ML-based tuning agents. The agents share an iterative workflow, following a trial-and-error manner to tune the DBMS. They utilize a ML model to predict a promising configuration (e.g., building an index), and evaluate the performance of the suggested configuration. Based on the evaluation result, they update the model to improve the efficacy for future decision making. Following the above paradigm, our framework defines an abstraction for the ML-based tuning agents. Concretely, we categorize the tuning agent based on the adopted algorithms and how they evaluate the suggested configuration, as discussed in Section 3.2. And, to support existing ML-based tuning agents, we define the following three base classes (BO, RL, and RLEstimator), and provide the corresponding interfaces, as shown in Table 1. We summarize the interfaces as follows.

- **InitModel**: initializes the model used by the tuning agent.
- **Suggest**: infers the defined model to predict a promising configuration.
- **UpdatePolicy**: updates the defined model using the input augmented observation.
- **UpdateEstimator**: updates the estimator using the input augmented observation.

This abstraction offers flexible support for different tuning algorithms, as it removes the heterogeneity issue in existing ML-based tuning agents. For each main class, the tuning agents differ in how to implement these interfaces of their logic (e.g., which kind of neural networks to adopt in **InitModel**, and how to predict a promising configuration based on their model in **Suggest**). Using these interfaces, it is easy to integrate an agent with existing implementation in UniTune for multi-component tuning. A user just needs to inherit the corresponding base class and overrides its functions. For example, to add a RL tuning agent – CDBTune [68], the user defines an actor-critic network for the agent in **InitModel**, and implement how to suggest configurations by inferring the actor network in **Suggest** as well as how to update the model based on the input observation in **PolicyUpdate**.

Table 1: Abstraction for ML-based tuning agents.

Type	Algorithm	Interfaces
BO	Bayesian Optimization	Suggest, ModleInit, UpdatePolicy
RL	Reinforcement Learning	Suggest, ModleInit, UpdatePolicy
RLEstimator	Reinforcement Learning	Suggest, ModleInit, UpdatePolicy, UpdateEstimator

[Tuning-Setting]

```
components = {'index': 'DBA-Bandit', 'knob': 'OtterTune',
              'query': 'LearnedRewrite'}
tuning_budget = 108000
performance_metric = 'execution-time'
```

Figure 3: An Example of User Interface.

Given the user-implemented interfaces, our framework coordinates the agents of different components automatically, hiding underlying details for users. First, for one agent, our framework automates its pipelined execution using these functions. For BO and RL, it evaluates the configuration output by **Suggest** by interacting with the DBMS and inputs the resulting observation to **PolicyUpdate**. For **RLEstimator**, it samples a configuration, evaluates it in the DBMS, and inputs the resulting observation to **UpdateEstimator**. Second, our framework schedules the agents to tune multiple components in a DBMS wisely by maintaining the running records of each agent. It decides how to allocate limited tuning budget among agents automatically (discussed in Section 6) and manages the agents following a message propagation protocol to enable collaboration among agents (discussed in Section 5).

4.2 User Interface

To launch a multi-component tuning task, a user only needs to specify the database settings and the tuning setting. The database setting refers to the connection information and the workload information. The tuning setting describes the types of DBMS components to be tuned, the corresponding agent, the performance metric, and the overall tuning budget. For ease of usage, we adopt configuration files to define a task. The following code in Figure 3 gives an example and we omit the database setting for space constraints. It defines a task that configures index, knobs, and query components via the ML-based tuning agents – DBA Bandit [52], OtterTune [6] and LearnedRewrite [72], respectively. The tuning budget is set to 30 hours, and the tuning objective is the execution time.

4.3 Workflow

Figure 4 presents the overview of UniTune. The critical components are emphasized to illustrate its internal workflow. Given the tuning agents and the task information defined by a user, the two modules (*budget allocation* and *message propagation*) execute iteratively. First, the *budget allocation* module selects a promising tuning agent and the selected agent is passed to the *message propagation* module. Then, the *message propagation* module arranges the input agent to

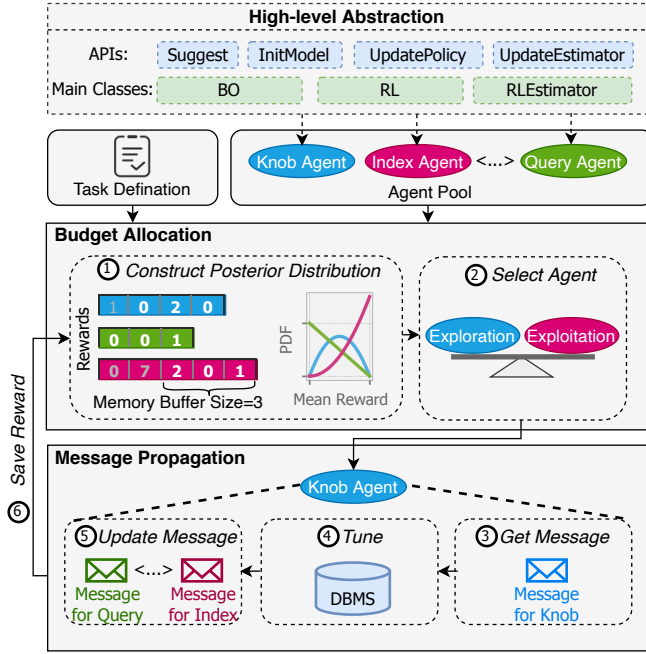


Figure 4: Workflow of UniTune. UniTune has a “high-level abstraction” which supports different ML-based tuning agents, a “budget allocation” module which selects promising tuning agents, and a “message propagation” protocol which enables the collaboration among agents to resolve their dependency.

tune the DBMS while cooperating with the other agents, and finally, the tuning reward obtained is returned to the *budget allocation* module to update the resource allocation strategy. The workflow ends once the tuning budget is spent out or a termination criterion is met. We explain the two modules as follows.

Budget Allocation selects a promising agent for tuning the DBMS at each step. The budget allocation problem can be formulated as a multi-arm bandit problem with non-stationary reward (discussed in Section 6.1). The goal is to maximize the cumulative expected reward (i.e., the improvement of a performance metric) within the given budget. The challenge is that the reward of an agent can only be observed after evaluating its actions using a certain budget, and the reward is non-stationary – it changes as the optimization proceeds. To make good decisions, this module judiciously balances exploration and exploitation via Thompson Sampling, a classic algorithm for online decision problems (discussed in Section 6.2). Concretely, it constructs the posterior distributions of reward for each agent based on the reward observations (step 1) and selects an agent based on the probabilities sampled from the posterior distribution (step 2). To adapt to the non-stationary reward, we use a memory buffer to constrain the number of considered reward observations when constructing the posterior distributions.

Message Propagation enables collaboration across tuning agents. The selected agent executes the procedure following a message propagation protocol, which specifies how it should process the

Algorithm 1 Top-level Design.

Parameter: Agents $\{A_i\}_{i=1}^m$, tuning budget K , sub-budget k .

Output: A suggested configuration for best performance.

```

1: function FRAMEWORK( $\{A_i\}_{i=1}^m, K$ )
2:   // Initialize global performance as the default one
3:    $perf_{global} \leftarrow \text{init}()$ .
4:   while tuning budget is available do
5:      $A_j \leftarrow \text{SelectAgent}(\{A_i\}_{i=1}^m)$ .
6:      $context \leftarrow \text{GetMessage}(A_j)$ .
7:     // Run the selected agent using a sub-budget
8:      $config_{inc}, f_{inc} \leftarrow A_j.\text{Run}(context, k)$ .
9:     // Save the performance improvement
10:     $A_j.\text{history.append}(\max(0, f_{inc} - f_{global}))$ .
11:    if  $f_{inc} \leq f_{global}$  then
12:       $f_{global} \leftarrow f_{inc}$ .
13:       $A_j.\text{apply}(config_{inc})$ .
14:      // Update messages to other agents
15:       $\text{UpdateMessage}(\{A_i\}_{i=1}^m - A_j)$ .
16:    end if
17:  end while
18: end function

```

received message and broadcast its tuning messages to other agents (discussed in Section 5). The agent first obtains its message which contains the tuning decisions made by other agents (step 3). Then, the agent encapsulates the message in its model (discussed in Section 5.2.2) and tunes its responsible component using a sub-budget (step 4). If it finds a better configuration, it applies the configuration to the DBMS and expresses its tuning decisions in the form of a context feature (discussed in Section 5.2.1), which is propagated to other agents (step 5). In the end, we append the observed reward to the corresponding record for later budget allocation (step 6).

Algorithm 1 gives the pseudo-code showing the main optimization loop of our framework. The sub-budget is a hyper-parameter that controls the amount of budget for running one agent. It should be larger than the maximal time for the evaluation in DBMS (e.g., stress testing) but as small enough to enable fine-grained budget allocation. In our implementations, we set it twice the timeout of stress testing time. UniTune first initializes the current best performance as the default performance (Line 3). Then it selects an agent (Line 5). The selected agent receives the message (i.e., a context feature) sent by other agents (Line 6), configures the DBMS using a sub-budget and returns the best configuration found during tuning and its performance (Line 8). UniTune saves the reward (i.e., performance improvement) (Line 10). If the selected agent finds a better configuration, it applies the configuration and update the message to other agents (Line 11-16).

5 MESSAGE PROPAGATION PROTOCOL

To enable collaboration across tuning agents, we propose a message propagation protocol to specify their behaviors. Within this protocol, the tuning agents execute alternately and broadcast their tuning decisions (i.e., messages) to other agents. In the following, we discuss our design options in Section 5.1 and introduce how to represent messages and how to make the agents learn from the

messages in Section 5.1. To support RL-estimator based agents with the protocol, we provide a new training paradigm in Section 5.3.

5.1 Design Options

The ML-based tuning agents use the evaluation in DBMS to obtain the performance of a configuration, as we analyzed. Therefore, to ensure the validation of the observation, the change of configurations (including the configurations of other components) can not occur during the evaluation so that we can not run two agents simultaneously. To this end, there are two options for tuning multiple components in a database system. (1) *joint optimization* that joins the subspaces of DBMS components and runs a single agent to optimize over the joint space, and (2) *alternating optimization* that runs multiple tuning agents alternatingly to optimize over the corresponding subspaces.

Our protocol adopts alternating optimization. The joint optimization has severe scalability issues and it relies on a single tuning agent which is hard to adapt to different DBMS components. First, the number of evaluations needed to reach the global optimum increases exponentially as the dimension of the search space grows. The presence of some regions with large posterior uncertainties can result in over-exploration and failure to exploit promising areas. Standard Bayesian optimization may perform worse than random search in some high-dimensional spaces [61]. And reinforcement learning suffers from sparse rewards, and may not learn a successful policy [62]. Second, the underlying objective function is complex, and thus fitting one global model over a huge space is difficult. For example, SmartI [47], a RL based agent for index selection, adopts Deep Q-Learning for discrete subspace, and CDBTune [68], a RL based agent for knob tuning, adopts DDPG for continuous subspace. It is non-trivial to jointly optimize in two subspaces by a single agent. In contrast, alternating optimization solves the optimization problem efficiently by decomposing the full space to subspaces, which is to only configure the selected components through the corresponding tuning agents. Thus, it can take full advantage of the emerging techniques for tuning a specific component. For example, we could run two agents alternatingly. First, we can apply, e.g., SmartI, by consuming a fraction of the budget, and fix the best indexes found so far to the DBMS. Second, we switch to, e.g., CDBTune, and use another fraction of the budget to figure out a best knob that the new agent could find out. In UniTune, the budget allocation module decides which agent to execute step by step, as discussed in Section 6. In each step, the alternating optimization essentially fixes some dimensions of the joint space and optimizes over a subspace, which is much smaller than the full space and can be optimized effectively. The process searches for the optimum by exploring different subspaces alternatingly.

5.2 Message Broadcast via Context

Existing tuning agents work under the assumption that the configurations of other components are fixed. In UniTune, if an agent finds a better configuration, it applies the configuration to its controlled component and fix the configuration when other agents tune the DBMS. However, if one agent modifies its component, the modeling of the other agent will be inaccurate due to environmental changes. Therefore, we need to broadcast its tuning decisions to

other agents, and thus the other agents can be aware of the environmental changes in time. Then, the next question is how to make the affected agents respond to the environmental changes properly. One possible response is to initialize their ML models and search from scratch in the new environment. However, the previous tuning history will be forgotten, and building a new model from scratch needs a large number of observations. Thus, we turn to another direction – we make the agents adapt to the environmental changes. Intuitively, the tuning policies in different environments share certain common knowledge. We refer to the environmental feature as context. There exists a mapping from context and configuration to the performance metric [71]. We expect the same configuration across correlated contexts to have similar performance metrics. Utilizing the correlations between contexts can significantly speed up the tuning process. To this end, we decide to express the messages broadcast among agents in the form of context features and explore two sub-questions: (1) how to characterize the context, and (2) how to make the agents learn across contexts.

5.2.1 Context Characterization. The most direct way is to concatenate all the configurations of other components as a context feature since it indicates the environment changes directly due to other agents' tuning behaviors. However, the configurations might be unstructured. For example, the agent for query rewrite applies several (operator, rule) transmissions to a query in the workload. The number of transmissions is mutative. Extra feature engineering effort is needed to support the extension of different agents. Inspired by OtterTune [6], we resort to the DBMS's internal runtime metrics to characterize the effect of configurations. All modern DBMSs expose a large amount of information about the system, such as statistics on the number of pages read/written, query cache utilization, and locking overhead. And they are affected by the configuration settings. To observe a context for an agent, we fix the best-ever configurations of other components, keep the configuration of its responsible components as default, conduct stress tests on the DBMS to collect the internal metric statistics.

5.2.2 Context Encapsulation. To utilize the tuning messages, we need to encapsulate the context feature in the models of tuning agents. For a BO based agent, we augment the context feature to the input of the surrogate. Then the surrogate models the mapping from context and configuration to the performance. For a RL based agent, we concatenate the context feature to the observed state. Then the tuning policies, a mapping from state to action, learned by the neural network will be similar among correlated contexts. For a RL-estimator based agent, we add the context feature to the input of the estimator. Then the estimator can estimate the utility of given actions in different contexts. In a summary, the format of the training data for RL agents is $\langle \text{state}, \text{configuration}, \text{next state}, \text{reward} \rangle$, where both "state" and "next state" here contain the concatenated context features. And the format for BO and RL-estimator agents is $\langle \text{context feature}, \text{configuration}, \text{performance} \rangle$.

5.3 Uncertainty-aware Training for RL-estimator Based Agents

In alternating optimization, if one tuning agent finds a better configuration, we apply it to the DBMS and broadcast its tuning decisions

to other agents. Then, other agents could update their suggestions based on the received message, moving towards a better configuration composition. It is straightforward for BO based and RL-based agents since they allocate sub-budget to update tuning policies and broadcast the tuning messages when they are invoked. However, for RL-estimator based agents, in the existing two-step manner, as shown in Figure 2c, they first use the allocated sub-budget to train an estimator by the expensive evaluation in DBMS (step 1). After the estimator is trained, the RL agent updates its tuning policy and only recommends a final configuration (step 2). Therefore, during step 1 (most of the time), the other agents lose the opportunity to refine their suggestions according to the tuning decisions of the RL-estimator agent. In addition, the distribution of randomly sampled configurations in step 1 may not coincide with the configurations inferred by the RL agent in step 2, leading to inaccurate estimation.

To address these issues, we design an uncertainty-aware training schema for RL-estimator based agents. We train the RL agent and estimator in one step, enabling the RL agent to suggest promising configurations every time the agent is invoked. Specifically, we adopt an uncertainty-aware surrogate to estimate the uncertainty of the configurations, such as the Gaussian citation. When a RL-estimator based agent is invoked, we first train the RL agent, inferring the estimator. And we put the inferred configuration in a priority queue with estimated uncertainty as the priority. After the RL agent converges, we evaluate its recommended configuration by interacting with the DBMS. If a better configuration is found, we apply it and broadcast it to other agents, facilitating cooperation among agents. Then we use the left sub-budget to evaluate the performance of the configurations in the priority queue and update the estimator based on the augmented observations. The uncertainty-aware schema guides the sampling of configurations in a demand-oriented way and leads to more accurate prediction from the estimator.

6 BUDGET ALLOCATION

Various policies can be employed to allocate a tuning budget among agents. One simple strategy is a round-robin that equally allocates the tuning budget in turns. However, the reward of different agents often varies dramatically. For example, some workloads are very sensitive to the index configuration while query rewrite would offer little or even no improvement. Therefore, we should spend more budget on learning good indexes instead of rewriting the queries. The round-robin would waste the budget on the agents which turn out to have little reward. To wisely allocate the budget, we find that agent selection can be defined as a multi-armed bandit (MAB) problem [9] from statistical machine learning, where different arms refer to the tuning agents. In the following, we discuss the formulation of this problem and present a Thompson Sampling based strategy with a memory buffer to solve the MAB.

6.1 A MAB View for Agent Selection

The MAB problem has been a subject of intense studies in statistics for decades. The name comes from imagining a gambler at a slot machine with multiple arms, who has to decide which arm to pull [9]. Each time an arm is pulled, the gambler receives a payout. Because the distribution of payouts corresponding to each arm is

not listed, the gambler can learn it only by experimenting. Agent selection problem shares a similar essence with MAB since they both sequentially make online decisions to maximize a total payoff from unknown distributions.

MAB for Agent Selection. We formally define the stochastic MAB problem in the agent selection case. We are given a set of tuning agents $\mathbf{A} = \{A_i\}_{i=1}^m$ and each agent is an “arm”. Let $\mathbf{T} = \{1, 2, \dots, T\}$ denotes a sequence of decision epochs (T can be derived from the setting of tuning budget and sub-budget). At each decision epoch, we must select an agent to execute. After selecting an agent $A_j \in \mathbf{A}$ at epoch $t \in \mathbf{T}$, a real-valued reward $X_t^j \in \mathbb{R}$ is observed, where X_t^j is a random variable with expectation $\mu_t^j = \mathbb{E}[X_t^j]$. The goal is to maximize the expected cumulative reward in epochs \mathbf{T} :

$$\arg \max_{j(1), j(2), \dots, j(T)} \mathbb{E} \left[\sum_{t=1}^T \mu_t^{j(t)} \right],$$

where $j(t)$ is the arm selected in step t .

Reward Function. Since we aim to find a configuration composition that optimizes the given performance metric, the semantics of the reward has to be consistent with the tuning goal. Therefore, we define the reward as the improvement of the performance metric when running the agent. Specifically, the reward is

$$X_t^j = \max \left(0, f_{t,inc}^j - f_{t,global} \right),$$

where $f_{t,inc}^j$ denotes the best performance achieved by agent A_j at epoch t , and $f_{t,global}$ denotes the best performance achieved before t , assuming that we want to maximize the performance metric f .

Technical Challenges: To maximize the cumulative reward, we face two challenges. The first is the common dilemma in the MAB problem – the trade-off between exploitation and exploration:

- **Exploitation:** one may want to allocate budget to the agents that already yielded high reward in the past.
- **Exploration:** one may also want to allocate budget to the agents that might earn higher reward in the future.

The second is a trade-off between “remembering” and “forgetting” caused by the non-stationary reward in the agent selection case. In the conventional MABs, the reward distributions do not change over time. However, as we analyzed, the reward distribution of an agent is not stationary – it changes as the optimization proceeds. In general, the reward of an agent decays since the performance improvement achieved by the agent tends to be saturated as the budget is consumed (i.e., the decreasing marginal returns). Therefore, the old experience might be no longer applicable to estimating future reward. Then, to select a promising agent, we should carefully balance the trade-off between remembering and forgetting:

- **Remembering:** one may want to keep track of more observations to decrease the variance of reward estimates.
- **Forgetting:** one may also want to dismiss “old” information which is less relevant due to possible changes in the underlying reward.

6.2 Thompson Sampling with Memory Buffer

To solve the above challenges, we propose a Thompson Sampling based strategy with a memory buffer. Thompson Sampling [53] is a classic reinforcement learning algorithm for the MAB problem. At a high level, it chooses an arm to play according to its probability

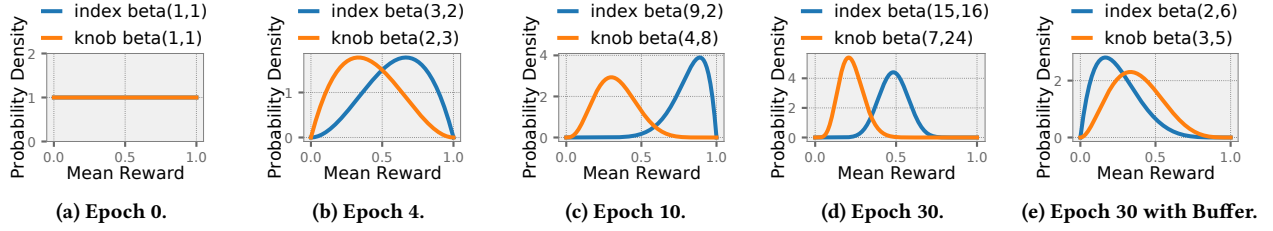


Figure 5: Posterior Distribution in Different Epochs. We present probability density functions over mean reward.

Algorithm 2 Agent Selection.

Parameter: Agents $\{A_i\}_{i=1}^m$, buffer size s , a scale factor $rfactor$.

Output: A selected agent.

```

1: function AGENTSELECTION( $\{A_i\}_{i=1}^m$ )
2:   for  $i \leftarrow 1, 2, \dots, m$  do
3:      $S \leftarrow 0, F \leftarrow 0$ 
4:     for  $reward \in A_i.history[-s:]$  do
5:       if  $reward > 0$  then
6:          $S \leftarrow S + \text{round}\left(\frac{reward}{rfactor}\right)$ 
7:       else
8:          $F \leftarrow F + 1$ 
9:       end if
10:    end for
11:    Sample  $w_i$  from the  $Beta(S + 1, F + 1)$  distribution.
12:  end for
13:  return  $\arg \max_{i=1}^m w_i$ .
14: end function

```

of being the best arm. It builds up experience (i.e., the observation of selected agents and their reward) to construct a posterior distribution of reward. And, at each epoch, the algorithm selects an arm according to its sampled probability of being the arm with the highest reward.

Exploitation & Exploration. Thompson Sampling addresses the first challenge by sampling the arm from the posterior distribution instead of greedily selecting the arm with the highest expected reward. Consider the example of running two agents – index agent and knob agent. Figure 5 presents their changing posterior distributions across epochs. These distributions represent the beliefs of the arms given the observed history. At epoch 0 (Figure 5a), we do not have any observation for the reward. Therefore, these two agents will be selected with the same probability. At epoch 4 (Figure 5b), the index agent has gained more reward. And a greedy policy will select the index agent since it has a larger expected reward. But in Thompson sampling, the knob agent can still be selected, since the selected arm is sampled from the posterior distribution. At epoch 10 (Figure 5c), the index agent is more likely to be selected, since the posterior is more differentiated with smaller uncertainty (but the knob agent still has the possibility to be selected by Thompson sampling). Intuitively, if we want to maximize exploration, one would choose the agent entirely at random. If we want to maximize exploitation, one would greedily choose the agent with the highest expected reward. Sampling from the posterior distributions achieves a balance between the two goals.

Remembering & Forgetting. However, recalling the second challenge that the reward distribution is non-stationary in our case, the conventional Thompson Sampling would be less effective. Continuing with the example in Figure 5, we assume that the reward of the index agent is saturated after epoch 10 (i.e., the index agent will consecutively fail to find better configurations). But given the large reward of the index agent in the first 10 epochs, its expected reward at epoch 30 (Figure 5d) is still larger than that of the knob agent with relatively small uncertainty. Therefore, the index agent is more likely to be selected although it offers no future reward. To address the second challenge, we adopt a simple yet effective strategy – we use a memory buffer to control the number of considered observations for constructing the posterior distribution. Given the observed reward $\{X_t^j\}_{t=0}^{T_j}$ for agent A_j , we only utilize $\{X_t^j\}_{t=T_j-s+1}^{T_j}$ (i.e., the previous s observations) to construct its posterior distribution, where s is a memory buffer size. The buffer size represents the number of considered observations. With a smaller buffer size, the algorithm will have a lower risk of being biased by past observations and can better adapt to the changes in future reward. But, at the same time, it may forget useful knowledge, missing the opportunity for exploitation. The buffer size controls the trade-off between forgetting and remembering. And it has been theoretically proved that sub-linear regret is achievable in non-stationary environments with an appropriate trade-off between forgetting and remembering [25]. Figure 5e presents the posterior distribution with a buffer size of seven. The expected reward of the knob agent is higher than that of the index agent and will be more likely to be selected since it gains more reward in the considered previous seven epochs.

As we discussed, there are dependencies among different tuning agents. The allocation strategy explores different dependency directions through different tuning orders of components. It decides the tuning orders step by step intrinsically by selecting a promising agent at each iteration. For example, UniTune might start by evaluating one tuning order, if the improvement is small, it explores other directions. Thus, different tuning orders are explored adaptively based on historic feedback. Algorithm 2 presents a formal description of the budget allocation strategy in UniTune. We adopt the beta distribution to describe the posterior distribution of reward, which is widely used in Thompson Sampling [5]. Beta distribution has two parameters, S and F to control the estimation of the expected reward. As shown in Figure 5, after pulling an arm, the posterior distribution of the expected reward can be constructed by simply adjusting the two parameters. And higher the S and F

are, the tighter the concentration of $Beta(S, F)$ is around the mean. To trade off between remembering and forgetting, we utilize the observed reward in a memory buffer with size s to update the two parameters (Line 2-8). Then, we sample from the posterior distributions (Line 11) and select an agent according to the probability of its mean being the largest (Line 13). The conventional Thompson Sampling observes binary reward and hence is not directly applicable to our case of continuous reward. “Probabilistic reward” [5] is used to adapt Thompson Sampling to the continuous reward scenario – it scales the continuous reward to a domain of $[0, 1]$ and samples a binary reward from a Bernoulli distribution with the scaled reward as its success probability. However, the “probabilistic reward” cause the agent with a positive observed reward possibly to have zero sampled reward. The information loss compromises the already sparse reward in our case. Based on our empirical experience, we scale the observed reward by a constant factor and use the rounded value to update S (Line 6).

7 EXPERIMENTAL EVALUATION

We conduct experiments to evaluate UniTune. We describe experimental setup in Section 7.1, conduct end-to-end comparisons with baselines in Section 6, analyze UniTune in Section 7.3, and present case studies in Section 7.4.

7.1 Experimental Setup

Agents. In most of our experiments, we tune three components in DBMSs – index, knobs, and SQL query. Unless stated otherwise, we adopt a BO based agent, OtterTune [6] for knobs tuning, a BO based agent, DBA-Bandit [52] for index selection, and a RL-estimator based agent, LearnedRewrite [72] for query rewrite. To showcase UniTune’s extendability, we replace OtterTune with other knobs tuning agents: CDBTune [68] and MySQLTuner [4]. We also add a RL-estimator based agent, AutoView [27] to tune view component.

Workloads. We consider three analytic benchmarks: JOB [40], TPC-H [1] with a scale factor 10 and TPC-DS [2] with a scale factor 50 since they are widely used in the evaluations for the tuning agents [27, 34, 42, 52, 71]. And most agents for query rewrite or view generation target at the OLAP scenarios. The JOB dataset represents a typical real database with unbalanced data distributions. We use its provided 113 queries in the workload. TPC-H and TPC-DS are both decision support benchmarks that model a real-world data warehousing environment. For TPC-H, some queries have costs orders of magnitude higher than the average value. Adding them in the workload makes the index selection problem much simpler [34] because an index that decreases the cost of at least one of these queries would always outperform indexes for other queries by orders of magnitude. Therefore, we adopt 12 TPC-H queries, which include a mix of common execution patterns (i.e., pipelines, jobs with multiple joins and filters, and groupby aggregations). We also add three badly-written queries from the synthetic query set used in LearnedRewrite [72] to TPC-H workload. For TPC-DS, we use its provided 99 queries as the workload.

Performance metric and configuration space. We optimize the total query execution time of a given workload. For each workload, we generate an index candidate set as the set of all potentially useful indexes (e.g., columns appearing in the predicates)[12] and

it serves as a starting point for index selection tool which picks a subset of these indexes. We only consider single-column indexes and generate 59 index candidates for JOB, 54 index candidates for TPC-H, and 237 index candidates for TPC-DS. We set a 1500 MB storage budget on built indexes for JOB and TPC-H and set a 20000 MB budget for TPC-DS since it has a larger dataset and much more complex schema. For the three workloads, we tune 50 knobs. We utilize the query rewrite rules in Calcite [8], as LearnedRewrite[72] stated. As for view generation, we consider 29 view candidates for JOB, following the view generation module in Autoview [27].

Baselines. The baselines used in the end-to-end evaluation are listed as follows:

- **Standalone Tuning** tunes a single component with a single agent, as previous studies did. We compare three standalone baselines: DBA-Bandit [52], LearnedRewrite [72] and OtterTune [6].
- **Sequential Tuning** is a native way to tune multiple components in a DBMS. Given a tuning budget, a DBA could allocate it equally to the tuning agents and run them in turn. We compare all the permutations for the execution order: i-q-k, i-k-q, q-i-k, q-k-i, k-i-q, and k-q-i. (i-k-q denotes index-knob-query, and so forth.)
- **UDO [59]** adopt a two-layer schema, separating tuning for the heavy and light parameters. In the outer layer, an agent for heavy parameters suggests and applies a heavy configuration. Then, in the inner layer, agents for light parameters iterate for a fixed number of inner iterations – searching suitable light configurations under the applied heavy configuration and evaluating them. The best-evaluated performance achieved in the inner loop is considered the performance of the heavy configuration and is used to update the outer agent. We adopt the implementation released by the authors [3]. For the fairness of comparison, we add LearnedRewrite as the agent for query rewrite, which is not currently supported by UDO. Following the paper [59], we consider the query agent as the inner agent, since it does not configure the physical structure of a DBMS. The query agent rewrites queries to reduce their execution time before tuning knobs. And we set the number for its inner iterations to three based on experiments. Therefore, the numbers of inner iterations are three for the query agent and five (the same as the released implementation [3]) for the knob agent on each index configuration.

Setting. We optimize a MySQL database deployed on a cloud ECS instance with 16 vCPU and 32GB RAM. We use the official MySQL default configuration as the initial configuration with no indexes and views built. We set the tuning budget to 30 hours and set a 10-minute timeout for running one query. For UniTune, we set the sub-budget to 20 minutes and use a memory buffer with size seven by default. We also vary the setting of buffer size, to test its robustness. When allocating the first nine sub-budgets (i.e., initialized phase), UniTune runs the agents in a round-robin manner to bootstrap the Thompson Sampling. To set r_{factor} for Algorithm 2, we observe the maximal reward of executing an agent (i.e., r_{max}) for a sub-budget in the initialized phase and set the r_{factor} as $\frac{r_{max}}{20}$, resulting in scaled reward approximately ranging from 0 to 20.

7.2 End-to-end Evaluation

We compare UniTune with the baselines. Figure 6 presents the best performances they achieved over time. And Figure 7 breaks

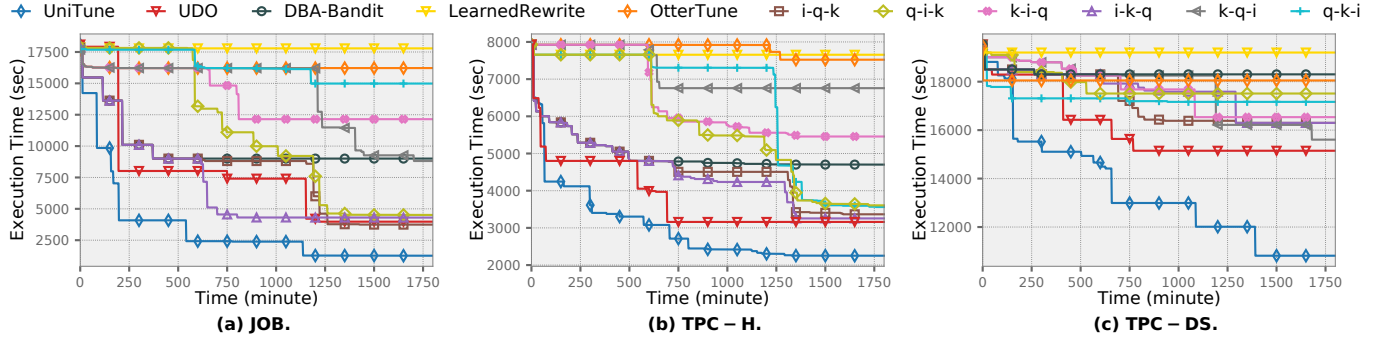


Figure 6: The best performance achieved over time by different baselines (bottom left is better).

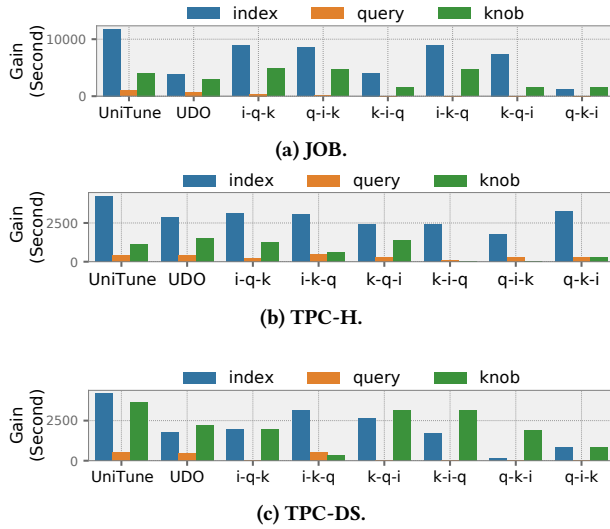


Figure 7: Performance breakdown in terms of tuning agents.

down the performance improvement of the three agents for multiple component tuning. We observe that UniTune finds the best configurations on both workloads, achieving $1.4 \times \sim 14.1 \times$ speedups on the execution time of the best configurations compared with baselines. And no matter how much the tuning budget (x-axis) is, UniTune outperforms the baseline approaches. The standalone tuning performs worse than the multiple components tuning in general, since they could only find sub-optimal configurations in a subspace. The six sequential approaches using different tuning orders converge differently. As shown in Figure 7, different orders lead to distinct improvements achieved by the three agents, indicating their dependencies. And considering the three workloads, there is no clear winner among the orders. But tuning indexes before knobs generally leads to better performance.

To deep dive, we analyze the baselines tuning multiple components, as shown in Figure 8. We observe that UniTune learns a proper allocation strategy by exploiting the historical observations while exploring the less-run agents. It allocates more budget on index and knob instead of query since tuning indexes and knobs offers more improvement compared with rewriting the queries for

the benchmark workload. And the number of configurations explored by UniTune on index, query, knob is approximately 4:2:3. In the first half period of tuning, UniTune learns to optimize more on the promising components (i.e., index), since it offers more tuning benefit, as shown in Figure 7. Then, it explores other agents in the second half period as the performance improvement of tuning indexes tends to become saturated. UDO outperforms sequential approaches in most cases, especially when the data size is larger (i.e., on TPC-H and TPC-DS). The index agent in UDO reorders the index configurations and largely reduces the reconfiguration overheads. However, UDO performs worse than UniTune. Its two-layer schema can not flexibly invest the tuning budget on promising agents and causes insufficient budget allocated on the outer agent (i.e., index agent), as shown in Figure 8(b). UDO applies one index configuration in the outer layer and executes the query agent for three inner iterations and the knob agent for five inner iterations. The ratio of configurations explored by UDO on index, query, knob is fixed to 1:3:5 during tuning, mismatching the situation that spending more budget on tuning indexes is beneficial.

7.3 Analysis of UniTune

We carefully design UniTune with message propagation protocol and a budget allocation strategy. In this section, we analyze UniTune's execution time, evaluate the corresponding designs via ablation study and variants comparison and then validate the robustness of UniTune on the setting of memory buffer size.

7.3.1 Execution Time Breakdown. The total execution time of UniTune contains the time for agent selection and the time for agent execution. The latter is controlled by the sub-budget during which the selected agent tunes the DBMS and updates its tuning policy. The former is negligible since it follows closed-form equations in Algorithm 2 without training ML models. It takes always less than 2 milliseconds in our experiments.

7.3.2 Ablation Study of Context Features. We encapsulate context features in the agents' models to learn the tuning policy. To validate the function of context features, we compare other solutions without context features, including (1) UniTune-w/o-C(reinit), which reinitializes an agent's model when its background environment changes (i.e., the other agents change the configuration of their components), as discussed in Section 5.2. (2) UniTune-w/o-C, which

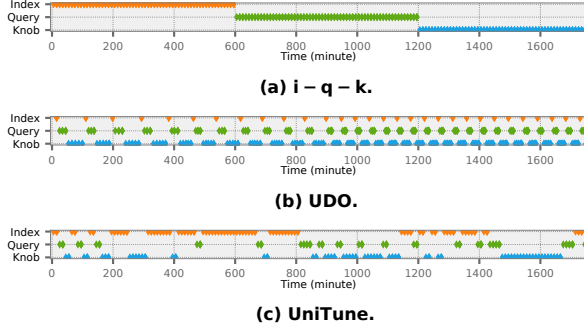


Figure 8: Budget allocation patterns of one sequential tuning approach (i-q-k), UDO, and UniTune on the index, query, and knob agents over time on JOB. (The color block indicates that the budget is allocated to the corresponding agent at the corresponding time slot.)

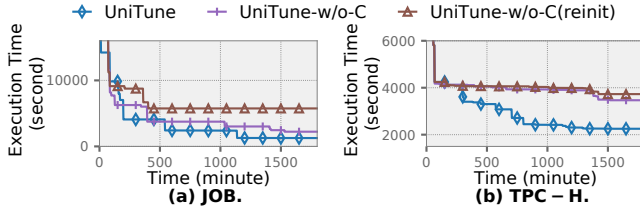


Figure 9: Ablation study of context features.

ignores the environmental changes. As shown in Figure 9, UniTune outperforms the solutions without context features. UniTune-w/o-C(reinit) performs the worst since its relearning method loses previous knowledge. Compared with UniTune-w/o-C, UniTune offers more advantages, indicating the relationship between configurations and performance has variations in different environments.

7.3.3 Comparison of Different Budget Allocation Strategies. We propose a Thompson Sampling based strategy with a memory buffer to allocate the tuning budgets among agents. We compare four allocation strategies: (1) TS-buffer, the Thompson Sampling based strategy with memory buffer (the one adopted in UniTune), (2) TS, which is the conventional Thompson Sampling, (3) Round-robin, which allocates the budget equally in an order of index-query-knob, as discussed in Section 6, (4) UCB, which adopts the contextual UCB algorithm [14]. It learns a mapping from context to the reward of an agent and selects the agents with the reward whose upper confidence bound is maximal. Figure 10 presents the comparison result and Figure 11 shows their budget allocation patterns. TS-buffer outperforms the other strategies, indicating it could allocate the tuning budget properly. It allocates more budget to the index agent in the first half period of tuning since configuring indexes gains more performance improvement, especially in the bootstrap phase. And the budget allocated to the index agent decreases in the second half period as the performance improvement of tuning indexes tends to become saturated. TS does not dismiss any outdated information and it allocates too much budget to the index agents in the second half period. Round-robin allocates the budget equally, wasting the budget on less promising agents (i.e., query

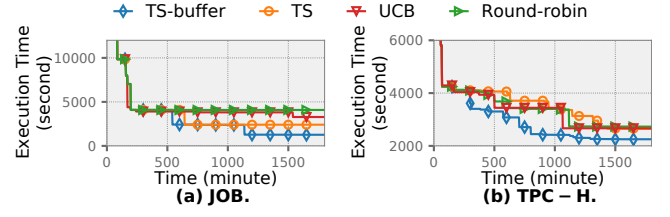


Figure 10: Comparison of different budget allocation strategies. TS-buffer is the strategy adopted in UniTune.

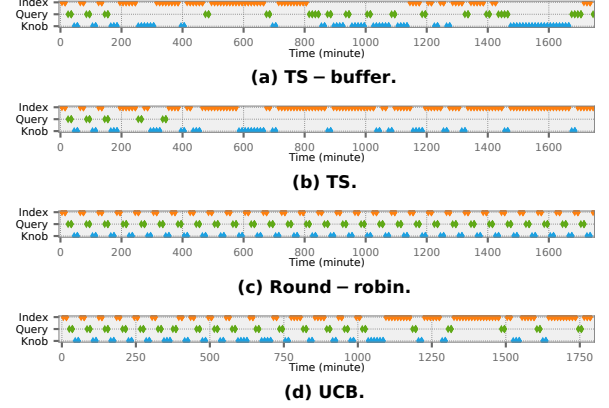


Figure 11: Budget allocation under different strategies for index, knob, and query agents over time on JOB.

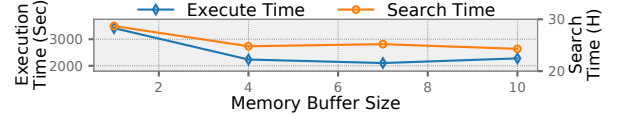


Figure 12: Impact of memory buffer size. We report the best performance (i.e. minimal execution time) and the search time to reach the best performance.

agent). UCB models the reward of selecting an agent in a given context. Compared with TS-buffer which directly selects the agent according to the observed rewards in a time window, learning the model requires more observations. Given the very limited observations (our case), UCB tends to explore the agents since the input contexts are unlikely to be covered by the previous observations.

7.3.4 Impact of Memory Buffer Size. UniTune restricts the number of considered observations for the reward of selecting an agent. While a smaller buffer size leads to a more timely response to reward changes, it may cause a loss of useful information. We evaluate UniTune with different memory buffer sizes and present their performance in Figure 12. The size of one causes inferior performance since lacking the exploitation of historical observations. In general, using buffer sizes from four to ten leads to good performance (seven is the default setting in our experiments).

7.4 Case Study

We use case studies to showcase the advantage of UniTune.

Table 2: Best performances achieved by different methods and their corresponding search time.

	Default	Grid search	UniTune	TS	Round-robin	UDO	i-q-k
Search Time (H)	/	934.4	2.6	3.2	3.8	3.1	4.6
Execution Time (Sec)	581.41	27.11	27.32	27.32	27.32	29.87	29.76

7.4.1 Comparison with Grid Search. To construct a ground-truth baseline, we define a small configuration space that can be enumerated by grid search. We optimize one query in TPC-H, Q19. It has 10 index candidates and 81 feasible index configurations under the 1500 MB storage budget. We tune three important knobs¹ selected based on the Gini score and we stipulate that each knob can only have three values, resulting in 63 feasible knob configurations². And there are four rewrite ways for Q19, not considering loop rewriting. To this end, we have 20412 ($81 \times 63 \times 4$) possible configurations in the configuration space, and we can evaluate their performances by a grid search to obtain the optimal configuration. Table 2 presents the performance of different baselines, where we only show the best sequential tuning method due to space constraints. Grid search finds the optimal configuration at the cost of an extremely long search time. UniTune finds the close-to-optimal configuration with the shortest search time. TS and round-robin can find the same configuration but with a longer search time.

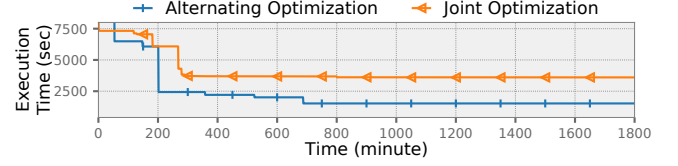
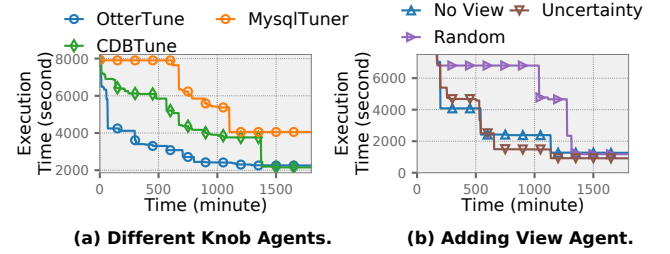
7.4.2 Comparison Between Joint Optimization and Alternating Optimization. UniTune executes the agent alternatingly. To tune multiple components in a DBMS, another solution is to adopt a central agent over the joint configuration space, as discussed in Section 5.1. The joint optimization has a scalability issue, due to the exponential growth of configuration space when joining the subspaces. We validate the analysis empirically. We adopt the two approaches to tune the knob and index components on JOB. For alternating optimization, we adopt two BO based agents. For joint optimization, we adopt one BO based agent optimizing over the joint configuration space. As shown in Figure 13, the alternating approach finds better configurations than the joint approach. Consider the knob agent tuning 50 knobs and index agent with 59 index candidates. For ease of analysis, we assume each knob has 10 possible values. Then the joint approach optimizes over a configuration space with a size of $10^{50} \times 2^{59}$, 10^{17} times larger than the summation of the two original subspaces. The alternating optimization addresses this scalability issue by decomposing the configuration space as the original agents assume and has a faster convergence speed.

7.4.3 Extendability. We integrate different tuning agents in UniTune to showcase its extendability.

Adopting Different Knob Agents. We still tune the three components: index, knob, and query in TPC-H. But, we adopt different knobs tuning agents respectively, OtterTune [6], CDBTune [68], and MySQLTuner [4] in UniTune. MySQLTuner is a rule-based tuning agent. It examines the DBMS metrics and uses heuristic rules to suggest knob configurations. Figure 14 presents the result. UniTune converges to similar performances when adopting the two

¹They are innodb_buffer_pool_size, innodb_log_file_size, innodb_thread_concurrency, and table_definition_cache, respectively.

²The left 18 infeasible configurations violate the rule that the combined size of ib_logfiles should be larger than $200 \text{ kB} \times \text{innodb_thread_concurrency}$, causing the MySQL database to be shutdown.

**Figure 13: Comparison of joint and alternating optimizations.****Figure 14: Extendability of UniTune. In Figure 14a, we adopt three knob agents respectively. In Figure 14b, we add an agent for view generation and compare the uncertainty aware sampling and random sampling discussed in Section 5.3.**

ML-based tuners. But the convergence when adopting CDBTune is slower since its RL agent requires more tuning budget to learn a great number of neural network parameters, which is consistent with the existing studies [7, 69]. When adopting MySQLTuner, the performance is inferior, since its limited heuristics fail to find good knob configuration. But this adoption reveals that UniTune could be extended to non-ML based agents, such as rule-based and cost-based agents. When a non-ML based agent is coordinated by UniTune, the budget allocation module treats it like the ML-based agents – the agent executes once UniTune allocates tuning budget on it. The main difference is that the non-ML based agents do not need the context feature to suggest configurations since they suggest configurations based on fixed rules or the cost estimation from database optimizer, which do not involve training the models.

Adding View Agent. We add an RL-estimator based agent Autoview [27] for view generation and tune four components in JOB and set the storage budget for materialized view to 500 MB. We compare the two strategies when training the estimator, as discussed in Section 5.3. As shown in Figure 14b, tuning the four components achieves better performance compared to tuning the three components (i.e., No View), since the materialized views save redundant computations among queries. And the uncertainty-aware sampling achieves better performance.

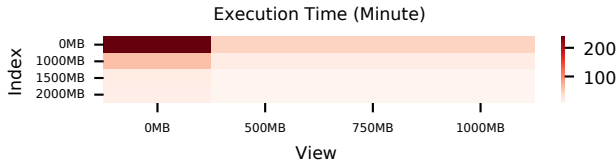


Figure 15: Impact of storage budget for index and view on the best tuning performance (i.e. minimal execution time).

7.4.4 Varying Storage Budgets. At the start of a tuning session, the storage budget is decided by users based on their affordable resource. In this experiment, we vary the storage budgets for index and view on JOB workload and present the best performance tuned by UniTune in Figure 15. We observe that the tuning performance gradually increases with the given storage budget and then saturates. For the view component, its tuning benefit saturates after being given 500MB storage budget, which is consistent with AutoView’s report [27]. For the index component, its tuning benefit increases significantly from zero storage budget to 1500 MB and then the increase becomes gentle. In our experiments, we use 1500MB for index storage budget and 500MB for view storage budget on the JOB workload.

8 CONCLUSION

In this paper, we reviewed the emerging studies on ML-based tuning agents in the database community and raised a question – “how to make them work together to configure multiple components in a DBMS?” To answer this, we proposed a unified and efficient coordinating framework UniTune for the ML-based tuning agents. We design a coordination protocol to enable collaboration among tuning agents and a strategy to allocate the tuning budget among the agents. We also define the interfaces adapted to a broad class of ML-based tuning agents, which are simple for integration with existing implementation and for future extension. We demonstrated that UniTune could support different ML-based agents and significantly outperforms the baselines.

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 author.

REFERENCES

- [1] 2015. TPC-H benchmark. <http://www.tpc.org/tpch/>.
- [2] 2015. TPC-H benchmark. <https://www.tpc.org/tpcds/>.
- [3] 2015. UDO Implementation. <https://github.com/jxiw/UDO>.
- [4] 2019. MySQL Tuning Primer Script. <https://github.com/major/MySQLTuner-perl>.
- [5] Shipra Agrawal and Navin Goyal. 2012. Analysis of Thompson Sampling for the Multi-armed Bandit Problem. In *COLT (JMLR Proceedings, Vol. 23)*. JMLR.org, 39.1–39.26.
- [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] 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.
- [8] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *SIGMOD Conference*. ACM, 221–230.
- [9] Sébastien Bubeck and Nicolò Cesa-Bianchi. 2012. Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems. *Found. Trends Mach. Learn.* 5, 1 (2012), 1–122.
- [10] 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.
- [11] 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.
- [12] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB’97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25–29, 1997, Athens, Greece*, Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jausfeld (Eds.). Morgan Kaufmann, 146–155. <http://www.vldb.org/conf/1997/P146.PDF>
- [13] Jin Chen, Guanyu Ye, Yan Zhao, Shuncheng Liu, Liwei Deng, Xu Chen, Rui Zhou, and Kai Zheng. 2022. Efficient Join Order Selection Learning with Graph-based Representation. In *KDD*. ACM, 97–107.
- [14] Wei Chu, Lihong Li, Lev Reyzin, and Robert E. Schapire. 2011. Contextual Bandits with Linear Payoff Functions. In *AISTATS (JMLR Proceedings, Vol. 15)*. JMLR.org, 208–214.
- [15] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *SIGMOD Conference*. ACM, 1241–1258.
- [16] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *Proc. VLDB Endow.* 2, 1 (2009), 1246–1257.
- [17] Gabriel Campero Durand, Marcus Pinnecke, Rufat Piriyev, Mahmoud Mohsen, David Briones, Gunter Saake, Maya S. Sekeran, Fabián Rodríguez, and Laxmi Balami. 2018. GridFormation: Towards Self-Driven Online Data Partitioning using Reinforcement Learning. In *aiDM@SIGMOD*. ACM, 1:1–1:7.
- [18] Gabriel Campero Durand, Rufat Piriyev, Marcus Pinnecke, David Briones, Balasubramanian Gurumurthy, and Gunter Saake. 2019. Automated Vertical Partitioning with Deep Reinforcement Learning. In *ADBIS (Short Papers and Workshops) (Communications in Computer and Information Science, Vol. 1064)*. Springer, 126–134.
- [19] Tamer Eldeeb, Zhengneng Chen, Asaf Cidon, and Junfeng Yang. 2022. Neuroshard: towards automatic multi-objective sharding with deep reinforcement learning. In *aiDM@SIGMOD*. ACM, 1:1–1:12.
- [20] 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.
- [21] Béatrice Finance and Georges Gardarin. 1991. A Rule-Based Query Rewriter in an Extensible DBMS. In *ICDE*. IEEE Computer Society, 248–256.
- [22] Jianling Gao, Nan Zhao, Ning Wang, Shuang Hao, and Haoyan Wu. 2022. Automatic Index Selection with Learned Cost Estimator. *Information Sciences* (2022).
- [23] Jia-Ke Ge, Yanfeng Chai, and Yunpeng Chai. 2021. WATuning: A Workload-Aware Tuning System with Attention-Based Deep Reinforcement Learning. *J. Comput. Sci. Technol.* 36, 4 (2021), 741–761.
- [24] Yaniv Gur, Dongsheng Yang, Frederik Stalschus, and Berthold Reinwald. 2021. Adaptive Multi-Model Reinforcement Learning for Online Database Tuning. In *EDBT*. OpenProceedings.org, 439–444.
- [25] Yonatan Gur, Assaf Zeevi, and Omar Besbes. 2014. Stochastic Multi-Armed-Bandit Problem with Non-stationary Rewards. In *NIPS*. 199–207.
- [26] Shuai Han, Mingxia Liu, and Jian-Zhong Li. 2022. Efficient Partitioning Method for Optimizing the Compression on Array Data. *J. Comput. Sci. Technol.* 37, 5 (2022), 1049–1067.
- [27] Yue Han, Guoliang Li, Haitao Yuan, and Ji Sun. 2021. An Autonomous Materialized View Management System with Deep Reinforcement Learning. In *ICDE*. IEEE, 2159–2164.
- [28] Jonas Heitz and Kurt Stockinger. 2019. Join Query Optimization with Deep Reinforcement Learning Algorithms. *CoRR* abs/1911.11689 (2019).
- [29] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2020. Learning a Partitioning Advisor for Cloud Databases. In *SIGMOD Conference*. ACM, 143–157.
- [30] 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).
- [31] Yoshiteru Ishihara and Masahito Shiba. 2020. Dynamic Configuration Tuning of Working Database Management Systems. In *LifeTech*. IEEE, 393–397.
- [32] Konstantinos Kanellis, Ramnathan Alagappan, and Shivaram Venkataraman. 2020. Too Many Knobs to Tune? Towards Faster Database Tuning by Pre-selecting Important Knobs. In *HotStorage*. USENIX Association.
- [33] 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.

- [34] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *Proc. VLDB Endow.* 13, 11 (2020), 2382–2395.
- [35] Jan Kossmann, Alexander Kastius, and Rainer Schlosser. 2022. SWIRL: Selection of Workload-aware Indexes using Reinforcement Learning. In *EDBT. OpenProceedings.org*, 2:155–2:168.
- [36] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *CoRR abs/1808.03196* (2018).
- [37] Mayuresh Kunjir and Shivnath Babu. 2020. Black or White? How to Develop an AutoTuner for Memory-based Analytics. In *SIGMOD Conference*. ACM, 1667–1683.
- [38] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2020. An Index Advisor Using Deep Reinforcement Learning. In *CIKM*. ACM, 2105–2108.
- [39] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2021. A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration. *Data Sci. Eng.* 6, 1 (2021), 86–101.
- [40] 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.
- [41] Guoliang Li, Xuanhe Zhou, and Lei Cao. 2021. Machine Learning for Databases. *Proc. VLDB Endow.* 14, 12 (2021), 3190–3193.
- [42] 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.
- [43] Yang Li, Jiawei Jiang, Jinyang Gao, Yingxia Shao, Ce Zhang, and Bin Cui. 2020. Efficient Automatic CASH via Rising Bandits. In *AAAI*. AAAI Press, 4763–4771.
- [44] Yang Li, Yu Shen, Huaijun Jiang, Tianyi Bai, Wentao Zhang, Ce Zhang, and Bin Cui. 2022. Transfer Learning based Search Space Design for Hyperparameter Tuning. In *KDD*. ACM, 967–977.
- [45] Yang Li, Yu Shen, Wentao Zhang, Jiawei Jiang, Yaliang Li, Bolin Ding, Jingren Zhou, Zhi Yang, Wentao Wu, Ce Zhang, and Bin Cui. 2021. VolcanoML: Speeding up End-to-End AutoML via Scalable Search Space Decomposition. *Proc. VLDB Endow.* 14, 11 (2021), 2167–2176.
- [46] Xi Liang, Aaron J. Elmore, and Sanjay Krishnan. 2019. Opportunistic View Materialization with Deep Reinforcement Learning. *CoRR abs/1903.01363* (2019).
- [47] Gabriel Paludo Licks, Júlia Mara Colleoni Couto, Priscilla de Fátima Mische, Renata De Paris, Duncan Dubugras A. Ruiz, and Felipe Meneguzzi. 2020. SmartIX: A database indexing agent based on reinforcement learning. *Appl. Intell.* 50, 8 (2020), 2575–2588.
- [48] Sijia Liu, Parikshit Ram, Deepak Vijaykeerthy, Djallel Bouneffouf, Gregory Bramble, Horst Samulowitz, Dakuo Wang, Andrew Conn, and Alexander G. Gray. 2020. An ADMM Based Framework for AutoML Pipeline Configuration. In *AAAI*. AAAI Press, 4892–4899.
- [49] Wei Lu, Xinyi Zhang, Zhiyu Shui, Zhe Peng, Xiao Zhang, Xiaoyong Du, Hao Huang, Xiaoyu Wang, Anqun Pan, and Haixiang Li. 2018. MSOL+: a Plugin Toolkit for Similarity Search under Metric Spaces in Distributed Relational Database Systems. *Proc. VLDB Endow.* 11, 12 (2018), 1970–1973.
- [50] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *aiDM@SIGMOD*. ACM, 3:1–3:4.
- [51] Andrew Pavlo, Matthew Butrovich, Ananya Joshi, Lin Ma, Prashanth Menon, Dana Van Aken, Lisa Lee, and Ruslan Salakhutdinov. 2019. External vs. Internal: An Essay on Machine Learning Agents for Autonomous Database Management Systems. *IEEE Data Eng. Bull.* 42, 2 (2019), 32–46.
- [52] R. Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic. 2021. DBA bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees. In *ICDE*. IEEE, 600–611.
- [53] Daniel Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. 2018. A Tutorial on Thompson Sampling. *Found. Trends Mach. Learn.* 11, 1 (2018), 1–96. <https://doi.org/10.1561/22000000070>
- [54] Zahra Sadri, Le Gruenwald, and Eleazar Leal. 2020. Online Index Selection Using Deep Reinforcement Learning for a Cluster Database. In *ICDE Workshops*. IEEE, 158–161.
- [55] Vishal Sharma and Curtis E. Dyreson. 2022. Indexer++: workload-aware online index tuning with transformers and reinforcement learning. In *SAC*. ACM, 372–380.
- [56] Vishal Sharma, Curtis E. Dyreson, and Nicholas Flann. 2021. MANTIS: Multiple Type and Attribute Index Selection using Deep Reinforcement Learning. In *IDEAS*. ACM, 56–64.
- [57] 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.
- [58] Danchi Wang, Chunyu Zhang, Wenbin Chen, Hui Yang, Min Zhang, and Alan Pak Tao Lau. 2022. A review of machine learning-based failure management in optical networks. *Sci. China Inf. Sci.* 65, 11 (2022).
- [59] Junxiong Wang, Immanuel Trummer, and Dehabrota Basu. 2021. UDO: Universal Database Optimization using Reinforcement Learning. *Proc. VLDB Endow.* 14, 13 (2021), 3402–3414.
- [60] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. WeTune: Automatic Discovery and Verification of Query Rewrite Rules. In *SIGMOD Conference*. ACM, 94–107.
- [61] Ziyu Wang, Masrour Zoghi, Frank Hutter, David Matheson, and Nando de Freitas. 2013. Bayesian Optimization in High Dimensions via Random Embeddings. In *IJCAI*. IJCAI/AAAI, 1778–1784.
- [62] Garrett Warnell, Nicholas R. Waytowich, Vernon Lawhern, and Peter Stone. 2018. Deep TAME: Interactive Agent Shaping in High-Dimensional State Spaces. In *AAAI*. AAAI Press, 1545–1554.
- [63] Sai Wu, Ying Li, Haoqi Zhu, Junbo Zhao, and Gang Chen. 2022. Dynamic Index Construction with Deep Reinforcement Learning. *Data Sci. Eng.* 7, 2 (2022), 87–101.
- [64] Wentao Wu, Chi Wang, Tarique Siddiqui, Junxiong Wang, Vivek R. Narasayya, Surajit Chaudhuri, and Philip A. Bernstein. 2022. Budget-aware Index Tuning with Reinforcement Learning. In *SIGMOD Conference*. ACM, 1528–1541.
- [65] Qiufen Xia, Lizhen Zhou, Wenhao Ren, and Yi Wang. 2022. Proactive and intelligent evaluation of big data queries in edge clouds with materialized views. *Comput. Networks* 203 (2022), 108664.
- [66] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *ICDE*. IEEE, 1297–1308.
- [67] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. 2020. Automatic View Generation with Deep Learning and Reinforcement Learning. In *ICDE*. IEEE, 1501–1512.
- [68] 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.
- [69] 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.
- [70] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuwei 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.
- [71] 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.
- [72] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A Learned Query Rewrite System using Monte Carlo Tree Search. *Proc. VLDB Endow.* 15, 1 (2021), 46–58.
- [73] Xuanhe Zhou, Luyang Liu, Wenbo Li, Lianyan Jin, Shifu Li, Tianqing Wang, and Jianhua Feng. 2022. AutoIndex: An Incremental Index Management System for Dynamic Workloads. In *ICDE*. IEEE, 2196–2208.
- [74] Jia Zou, Amitabh Das, Pratik Barhate, Arun Iyengar, Binhang Yuan, Dimitrije Jankov, and Chris Jermaine. 2021. Lachesis: Automated Partitioning for UDF-Centric Analytics. *Proc. VLDB Endow.* 14, 8 (2021), 1262–1275.