An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning

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[§]
[§]Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, China
[‡]Tsinghua University, China, [†]Tencent Inc., China
{jizhang, liu_yu, k.zhou, ytwbruce, vic, lillian_hust, mwran, zekangli}@hust.edu.cn
liguoliang@tsinghua.edu.cn; {tomxiao, bencheng, flacroxing}@tencent.com

ABSTRACT

Configuration tuning is vital to optimize the performance of database management system (DBMS). It becomes more tedious and urgent for cloud databases (CDB) due to the diverse database instances and query workloads, which make the database administrator (DBA) incompetent. Although there are some studies on automatic DBMS configuration tuning, they have several limitations. Firstly, they adopt a pipelined learning model but cannot optimize the overall performance in an end-to-end manner. Secondly, they rely on large-scale high-quality training samples which are hard to obtain. Thirdly, there are a large number of knobs that are in continuous space and have unseen dependencies, and they cannot recommend reasonable configurations in such high-dimensional continuous space. Lastly, in cloud environment, they can hardly cope with the changes of hardware configurations and workloads, and have poor adaptability.

To address these challenges, we design an end-to-end automatic CDB tuning system, CDBTune, using deep reinforcement learning (RL). CDBTune utilizes the deep deterministic policy gradient method to find the optimal configurations in high-dimensional continuous space. CDBTune adopts a try-and-error strategy to learn knob settings with a limited number of samples to accomplish the initial training, which alleviates the difficulty of collecting massive high-quality samples. CDBTune adopts the reward-feedback mechanism in RL instead of traditional regression, which enables end-to-end learning and accelerates the convergence speed of our model and improves efficiency of online tuning. We conducted extensive experiments under 6 different workloads on real cloud databases to demonstrate the superiority of CDBTune. Experimental results showed that CDBTune had a good adaptability and significantly outperformed the state-of-the-art tuning tools and DBA experts.

ACM Reference Format:
Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3300085

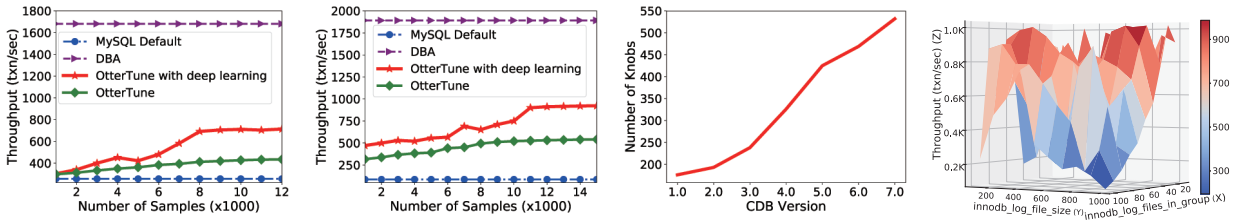
1 INTRODUCTION

The performance of database management systems (DBMSs) relies on hundreds of tunable configuration knobs. Superior knob settings can improve the performance for DBMSs (e.g., higher throughput and lower latency). However, only a few experienced database administrators (DBAs) master the skills of setting appropriate knob configurations. In cloud databases (CDB), however, even the most experienced DBAs cannot solve most of the tuning problems. Consequently, cloud database service providers are facing a challenge that they have to tune cloud database systems for a large number of users with limited and expensive DBA experts. As a result, developing effective systems to accomplish automatic parameters configuration and optimization becomes an indispensable way to overcome this challenge.

There are two classes of representative studies in DBMS configuration tuning: search-based methods [55] and learning-based methods [4, 14, 35]. The search-based methods, e.g., BestConfig [55], search the optimal parameters based on certain given principles. However, they have two limitations. Firstly, they spend a great amount of time on searching the optimal configurations. Secondly, they restart the search processing whenever a new tuning request comes, and thus fail to utilize knowledge gained from previous tuning efforts.

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3300085



(a) CDB (TPC-H) (b) CDB (Sysbench) (c) Knobs Increase (d) Performance surface

Figure 1: (a) and (b) show the performance of OtterTune [4] and OtterTune with deep learning over number of samples compared with default settings (MySQL v5.6) and configurations generated by experienced DBAs on CDB[1](developed by company Tencent). (c) shows the number of tunable knobs provided by CDB in different versions. (d) shows the performance surface of CDB (Read-Write workload of Sysbench, physical memory = 8GB, disk = 100GB).

The learning-based methods, e.g., OtterTune [4], utilize machine-learning techniques to collect, process and analyze knobs and recommend possible settings by learning DBA's experiences from historical data. However, they have four limitations. Firstly, they adopt a pipelined learning model, which suffers from a severe problem that the optimal solution of the previous stage cannot guarantee the optimal solution in the latter stage and different stages of the model may not work well with each other. Thus they cannot optimize the overall performance in an end-to-end manner. Secondly, they rely on large-scale high-quality training samples, which are hard to obtain. For example, the performance of cloud databases is affected by various factors such as memory size, disk capacity, workload, CPU model and database type. It is hard to reproduce all conditions and accumulate high-quality samples. As shown in Figures 1(a) and 1(b), without high-quality samples, OtterTune [4] or OtterTune with deep learning (we reproduce OtterTune and improve its pipelined model using deep learning) can hardly gain higher performance even though provided with an increasing number of samples. Thirdly, in practice there are a large number of knobs as shown in Figure 1(c). They cannot optimize the knob settings in high-dimensional continuous space by just using regression method like the Gaussian Process (GP) regression OtterTune used, because the DBMS configuration tuning problem that aims to find the optimal solution in continuous space is *NP*-hard [4]. Moreover, the knobs are in continuous space and have unseen dependencies. As shown in Figure 1(d), due to nonlinear correlations and dependencies between knobs, the performance will not monotonically change in any direction. Besides, there exist countless combinations of knobs because of the continuous tunable parameter space, making it tricky to find the optimal solution. Lastly, in cloud environment, due to the flexibility of cloud, users often change the hardware configuration, such as adjusting the memory size and disk capacity. According to statistics from Tencent, 1,800 users have made 6,700 adjustments in half a year. In this case, conventional machine learning have poor adaptability which needs to retrain the model to adapt to the new environment.

In this paper, we design an end-to-end automatic cloud database tuning system CDBTune using deep reinforcement learning (RL). CDBTune uses the reward functions in RL to provide a feedback for evaluating the performance of cloud database, and propose an end-to-end learning model based on the feedback mechanism. The end-to-end design improves the efficiency and maintainability of the system. CDBTune adopts a try-and-error method to enable utilizing a few samples to tune knobs for achieving higher performance, which alleviates the burden of collecting too many samples in initial stage of modeling and is more in line with the DBA's judgements and tuning action in real scenarios. CDBTune utilizes deep deterministic policy gradient method to find the optimal configurations in continuous space, which solves the problem of quantization loss caused by regression in existing methods. We conducted extensive experiments under 6 different workloads on four types of databases. Our experimental results demonstrated that CDBTune can recommend knob settings that greatly improve performance with higher throughput and lower latency compared with existing tuning tools and DBA experts. Besides, CDBTune has a good adaptability so that the performance of CDB deployed on configurations recommended by CDBTune will not decline even though the environment (e.g., memory, disk, workloads) changes. Note that some other ML solutions can be explored to improve the database tuning performance further.

In this paper, we make the following contributions:
(1) To the best of our knowledge, this is the first end-to-end automatic database tuning system that uses deep RL to learn and recommend configurations for databases.
(2) We adopt a try-and-error manner in RL to learn the best knob settings with a limited number of samples.
(3) We design an effective reward function in RL, which enables an end-to-end tuning system, accelerates the convergence speed of our model, and improves tuning efficiency.
(4) CDBTune utilizes the deep deterministic policy gradient method to find the optimal configurations in high-dimensional continuous space.
(5) Experimental results demonstrate that CDBTune with a good adaptability could recommend knob settings that greatly

[1]https://intl.cloud.tencent.com

improved performance and compared with the state-of-the-art tuning tools and DBA experts. Our system is open-sourced and publicly available on Github[2].

2 SYSTEM OVERVIEW

In this section, we present our end-to-end automatic cloud database tuning system CDBTune using deep RL. We firstly introduce the working mechanism of CDBTune (Section 2.1) and then present the architecture of CDBTune (Section 2.2).

2.1 CDBTune Working Mechanism

CDBTune firstly trains a model based on some training data. Then given an online tuning request, CDBTune utilizes the model to recommend knob settings. CDBTune also updates the model by taking the tuning request as training data.

2.1.1 Offline Training. We first briefly introduce the basic idea of the training model (and more details will be discussed in Sections 3 and 4) and then present how to collect the training data.

Training Data. The training data is a set of training quadruples $\langle q, a, s, r \rangle$, where q is a set of query workloads (i.e., SQL queries), a is a set of knobs as well as their values when processing q, s is the database state (which is a set of 63 metrics) when processing q, r is the performance when processing q (including throughput and latency). We use the SQL command "show status" to get the state s, which is a common command that DBA uses to understand the state of database. The state metrics keep the statistic information of the CDB, which describe the current state of database and are called *internal metrics*. There are 63 internal metrics in CDB, including 14 state values and 49 cumulative values. Example state metrics include buffer size, page size, etc., and cumulative values include data reads, lock timeouts, buffer pool in pages, buffer pool read/write requests, lock time, etc. All the collected metrics and knobs data will be stored in the memory pool (see Section 2.2.4).

Training Model. Because the DBMS configuration tuning problem that aims to find the optimal solution in continuous space is *NP*-hard [4], we use deep RL as the training model. RL adopts a try-and-error strategy to train the model, which explores more optimal configurations that the DBA never tried, reducing the possibility of falling in local optimum. Note the RL model is trained once offline which will be used to tune the database knobs for each tuning request from database users. The details of the training model will be introduced in Section 3.

Training Data Generation. There are two ways to collect the training data. (1) ***Cold Start.*** Because of lacking historical experience data at the beginning of the offline training process, we utilize standard workload testing tools (such as Sysbench) to generate a set of query workloads. Then for each query workload q, we execute it on CDB and get the initial quadruple. Then we use the above try-and-error strategy to train the quadruple and get more training data. (2) ***Incremental Training.*** During the later practical use of CDBTune, for each user tuning request, our system continuously gains feedback information from the user request according to configurations CDBTune recommends. With adding more real user behavior data to the training process gradually, CDBTune will further strengthen the model and improve the recommendation accuracy of the model.

2.1.2 Online Tuning. If a user wants to tune her database, she just needs to submit a tuning request to CDBTune which is in line with existing tuning tools like OtterTune and BestConfig. Once receiving an online tuning request from a user, CDBTune collects the query workload q from the user in recent about 150 seconds, gets the current knob configuration a, and executes the query workload in CDB to generate the current state s and performance r. Next it uses the model obtained by offline training to do online tuning. Eventually, those knobs corresponding to the best performance in online tuning will be recommended to the user. If the tuning process terminates, we also need to update deep RL model and the memory pool. The reason why we update the memory pool is that the samples produced by RL are generally sequential (such as configuration tuning step by step), which does not conform to the i.i.d. hypothesis between samples in deep learning. Based on this, we will randomly extract some batches of samples each time and update the model in order to eliminate the correlations between samples. Note that the user's workload of the database in online tuning process is different from the standard workload stored in the database. Hence, CDBTune needs to fine-tune the pre-training model in order to adapt to the real workload. There are mainly two differences between online tuning and offline training. On one hand, we no longer utilize the simulated data. Instead, we replay the user's current workload (see Section 2.2.1) to conduct stress testing on CDB in order to fine-tune the model. On the other hand, the tuning terminates if the user obtains a satisfied performance with the improvements over the initial configuration or the number of tuning steps reaches the predefined maximum. In our experiment, we set the maximum number to 5.

2.2 System Architecture

Figure 2 illustrates the architecture of CDBTune. The dotted box on the left represents the client, where users send their own requests to the server through the local interface. The other dotted box represents our tuning system in cloud, in which the controller under distributed cloud platform interacts information among the client, CDB and CDBTune. When

[2]https://github.com/HustAIsGroup/CDBTune

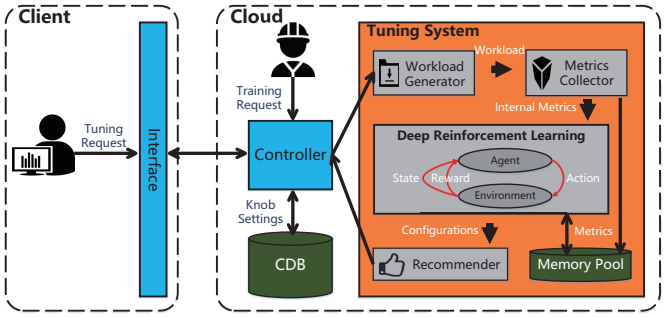


Figure 2: System Architecture.

the user initiates a tuning request or the DBA initiates a training request via the controller, the workload generator conducts stress testing on CDB's instances which remain to be tuned by simulating workloads or replaying the user's workloads. At the same time, the metrics collector collects and processes related metrics. The processed data will be stored in the memory pool and fed into the deep RL network respectively. Finally, the recommender outputs the knob configurations which will be deployed on CDB.

2.2.1 Workload Generator. From the description above, the workload generator we designed mainly completes two tasks: generating the standard workload testing and replaying the current user's real workload. Due to lacking samples (historical experience data) in initial training, we can utilize standard workload testing tools such as Sysbench[3]/TPC-MySQL[4] combined with the try-and-error manner of RL to collect simulated data, avoiding highly relying on real data. In this way, a standard (pre-training) model is established. When having accumulated a certain amount of feedback data from the user and recommending configurations to the user, we use the replay mechanism of the workload generator to collect the user's SQL records in a period of time and then execute them under the same environment so as to restore the user's real behavior data. By this means, the model can grasp the real state of the user's database instances more accurately and further recommend more superior configurations.

2.2.2 Metrics Collector. When tuning CDB upon a tuning request, we will collect and process the metrics data which can capture more aspects of CDB runtime behavior in a certain time interval. Since the 63 metrics represent the current state of database and are fed to the deep RL model in the form of vectors, we need to provide simple instructions when using them. For example, we take the average value of state value in a certain time interval and compute the difference between cumulative value at the same time. As for external metrics (latency and throughput), we take samples every 5 seconds and then simply calculate the mean value of sampled results to calculate reward (which represents how the performance of current database will change after performing corresponding knobs in Section 4.2). Note that the DBA also collects the average value for these metrics by executing the "show status" command during the tuning tasks. Although these values may change over time, the average value can well describe the database state. This method is intuitive and simple, and the experiment also validates its effectiveness. We also try other methods. For example, we replace the average value by taking the peak and trough values of metrics in a period of time, which just grasp the local state of database. Experimental results show that the peak and trough values are not as good as the average value due to lack of accurate grasp of the database state. Last but not least, as we use the "show status" command to get the database states, it will not affect the deployment under different workloads, environments and settings.

2.2.3 Recommender. When the deep RL model outputs the recommended configurations, the recommender generates corresponding execution setting parameter commands and sends the request of modifying configurations to the controller. After acquiring the DBA's or user's license, the controller deploys above configurations on CDB's instances.

2.2.4 Memory Pool. As mentioned above, we use the memory pool to store the training samples. Generally speaking, each experience sample contains four categories of information: the state of current database s_t (vectorized internal metrics), the reward value r_t calculated by reward function (which will be introduced in Section 4.2) via external metrics, knobs of the database to be executed a_t, and the database's state vector after executing the configurations s_{t+1}. A sample can be written as (s_t, r_t, a_t, s_{t+1}), which is called a transition. Like the DBA's brain, it constantly accumulates data and replay experience, so we call it experience replay memory.

3 RL IN CDBTUNE

To simulate the try-and-error method that the DBA adopts and overcome the shortcoming caused by regression, we introduce RL which originates from the method of try-and-error in animal learning psychology and is a key technology to solve *NP*-hard problems of database tuning in continuous space. All the notations used are listed in Appendix A.

3.1 Basic Idea

Both the search-based approach and the multistep learning-based approach suffer from some limitations, so we desire to design an efficient end-to-end tuning system. At the same time, we hope that our model can learn well with limited samples in the initial training and simulate the DBA's train of thought as much as possible. Therefore, we tried the

[3]https://github.com/akopytov/sysbench
[4]https://github.com/Percona-Lab/tpcc-mysql

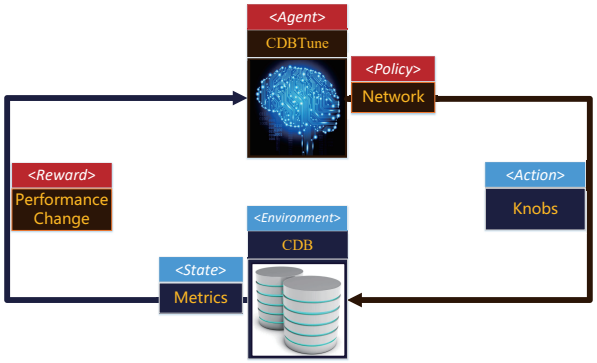


Figure 3: The correspondence between RL elements and CDB configuration tuning.

RL method. At the beginning, we tried the most classic Q-learning and DQN models in RL, but both of these methods failed to solve the problems of high-dimensional space (database state, knobs combination) and continuous action (continuous knobs). Eventually, we adopt the policy-based Deep Deterministic Policy Gradient approach which overcomes above shortcomings effectively. In addition, as the soul of RL, the design of reward function (RF) is vital, which directly affects the efficiency and quality of the model. Thus, by simulating the DBA's tuning experience, we design a reward function which is more in line with tuning scenarios and makes our algorithm perform effectively and efficiently.

3.2 RL for CDBTune

The main challenge of using RL in CDBTune is to map database tuning scenarios to appropriate actions in RL. In Figure 3, we describe the interaction diagram of the six key elements in RL and show the correspondence between the six elements and database configuration tuning.

Agent: Agent can be seen as the tuning system CDBTune which receives reward (i.e., the performance change) and state from CDB and updates the policy to guide how to adjust the knobs for getting higher reward (higher performance).

Environment: Environment is the tuning target, specifically an instance of CDB.

State: State means the current state of the agent, i.e., the 63 metrics. Specifically, when the CDBTune recommends a set of knob settings and CDB performs them, the internal metrics (such as counters for pages read to or written from disk collected within a period of time) represent the current state of CDB. In general, we describe the state at time t as s_t.

Reward: Reward is a scalar described as r_t which means the difference between the performance at time t and that at $t-1$ or the initial settings, i.e., the performance change after/before CDB performed the new knob configurations that CDBTune recommended at time t.

Action: Action comes from the space of knob configurations, which is often described as a_t. Action here corresponds to a knob tuning operation. CDB performs the corresponding action according to the newest policy under the corresponding state of CDB. Note that an action is to increase or decrease all tunable knobs values at a time.

Policy: Policy $\mu(s_t)$ defines the behavior of CDBTune in certain specific time and environment, which is a mapping from state to action. In other words, given a CDB state, if an action (i.e., a knob tuning) is called, the policy keeps the next state by applying the action on the original state. The policy here is the deep neural network, which keeps the input (database state), output (knobs), and transitions among different states. The goal of RL is to learn the best policy. We will introduce the details of deep neural network in Section 4.

RL Working Process. The learning process of DBMS configuration tuning in RL is summarized as follows. CDB is the target that we need to tune, which can be regarded as the environment in RL, while the deep RL model in CDBTune is considered to be the agent in RL, which is mainly composed of deep neural network (policy) whose input is the database state and output is the recommended configurations corresponding to the state. When executing the recommended configurations on CDB, the current state of database will change, which is reflected in the metrics. Internal metrics can be used to measure the runtime behavior of database corresponding to the state in RL, while external metrics can evaluate the performance of database for calculating the corresponding feedback reward value in RL. Agent will update its network (policy) according to these two feedback information to recommend superior knobs. This process iterates until the model converges. Ultimately, the most appropriate knob settings will be exposed.

3.3 RL for Tuning

RL makes a policy decision through the interaction process between agent and environment. Different from supervised learning and unsupervised learning, RL depends on accumulated reward, rather than labels, to perform training and learning. The goal of RL is to optimize its own policy based on the reward of environment through the interaction with environment and then achieve higher reward by acting on the updated policy. The agent is able to discover the best action through try-and-error manner by either exploiting current knowledge or exploring unknown states. The learning of our model follows two rules that the action depends on the policy, and the policy is driven by the expected rewards of each state. RL can be divided into two categories: value-based method and policy-based method. The output of the value-based method is the value or benefit (generally referred to as Q-value) of all actions and it chooses the action corresponding to the highest value. Differently, the output of the policy-based method is directly a policy instead of a

value and we can immediately output the action according to the policy. Since we need to use the actions, we adopt the policy-based method.

Q-Learning. It is worth noting that Q-Learning [30] is one of the most classic value-based RL methods, whose core is the calculation of Q-tables, which are defined as $Q(s, a)$. The rows of Q-table represent Q-value of the state while the columns of Q-table represent action, which measures how good it will be if the current state takes this action. The iterative formula of $Q(s, a)$ is defined below:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r + \gamma max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \quad (1)$$

The basis for updating Q-table is the Bellman Equation. In Eq. (1), α is the learning rate, γ is a discount factor, which pays more attention to short-term reward if close to zero and concentrates more on long-term reward when approaching one, and r is the performance at time $t + 1$.

Q-Learning is effective in a relatively small state space. However, it is hard to solve the problem of a large state space such as AlphaGo which contains as many as 10^{172} states, because a Q-table can hardly store so many states. In addition, the states of database in cloud environment are also complex and diverse. For example, suppose that each inner metric value ranges from 0 to 100 and its value is discreted into 100 equal parts. Then 63 metrics will have 100^{63} states. As a result, applying Q-Learning to database configuration tuning is impractical.

DQN. Fortunately, the proposed Deep Q Network (DQN) [32] method is able to solve the problem mentioned above effectively. DQN uses neural networks rather than Q-tables to evaluate Q-value, which fundamentally differs from Q-Learning (see Appendix B.3). In DQN, the input are states while the output are Q-value of all actions. Nevertheless, DQN still adopts Q-Learning to update Q-value, so we can describe the relationship between them as follows:

$$Q(s, a, \omega) \rightarrow Q(s, a)$$

where ω of $Q(s, a, \omega)$ represents the weights of neural network in DQN.

Unfortunately, DQN is a discrete-oriented control algorithm, which means the actions of output are discrete. Taking the maze game for example, only four directions of output can be controlled. However, knob combinations in database are high-dimensional and the values for many of them are continuous. For instance, if we use 266 continuous knobs which range from 0 to 100. If each knob range is discretized into 100 intervals, there will be 100 values for each knob. Thus there are totally 100^{266} actions (knob combinations) in DQN. Further, once the number of knobs increases or the interval decreases, the scale of outputs will increase exponentially. Thus, neither Q-Learning nor DQN can solve the issue of database tuning task [5]. Thus we introduce a policy-based RL method DDPG to address this issue in Section 4.

4 DDPG FOR CDBTUNE

Deep deterministic policy gradient (DDPG) [29] algorithm is a policy-based method. DDPG is the combination of DQN and actor-critic algorithm, and can directly learn the policy. In other words, DDPG is able to immediately acquire the specific value of the current continuous action according to the current state instead of computing and storing the corresponding Q-values for all actions, like DQN. Therefore, DDPG can learn the policy with high-dimensional states and actions, specifically, internal metrics and knob configurations. As a result, we choose DDPG in CDBTune to tune the knob settings. In this section, we first introduce a policy-based RL method DDPG, then describe our designed reward function, and finally sum up the advantages of applying RL to the database tuning problem.

4.1 Deep Deterministic Policy Gradient

We describe DDPG for CDBTune in Figure 4. When utilizing DDPG in CDBTune, firstly, we regard the CDB's instance which remains to be tuned as an environment E, and our tuning agent can obtain normalized internal metrics s_t from E at time t. Then our tuning agent generates the knob settings a_t and will receive a reward r_t after deploying a_t on the instance. Similar to most policy gradient methods, DDPG has a parameterized policy function $a_t = \mu(s_t|\theta^\mu)$ (θ^μ, mapping the state s_t to the value of action a_t which is usually called actor). Critic function $Q(s_t, a_t|\theta^Q)$ (θ^Q is learnable parameters) of the network aims to represent the value (score) with specific action a_t and state s_t, which guides the learning of actor. Specifically, critic function helps to evaluate the knob settings generated by the actor according to the current state of the instance. Inheriting the insights from Bellman Equation and DQN, the expected $Q(s, a)$ is defined as:

$$Q^\mu(s, a) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r(s_t, a_t) + \gamma Q^\mu(s_{t+1}, \mu(s_{t+1}))] \quad (2)$$

where the policy $\mu(s)$ is deterministic, s_{t+1} is the next state, $r_t = r(s_t, a_t)$ is the reward function, and γ is a discount factor which denotes the importance of the future reward relative to the current reward. When parameterized by θ^Q, the critic will be represented as $Q^\mu(s, a|\theta^Q)$ under the policy μ. After sampling transitions (s_t, r_t, a_t, s_{t+1}) from the reply memory, we apply Q-learning algorithm and minimize the training objective:

$$\min L(\theta^Q) = \mathbb{E}[(Q(s, a|\theta^Q) - y)^2] \quad (3)$$

where

$$y = r(s_t, a_t) + \gamma Q^\mu(s_{t+1}, \mu(s_{t+1})|\theta^Q)$$

[5] It is interesting to study how to wisely discretize the knobs to enable Q-learning or DQN to support this problem.

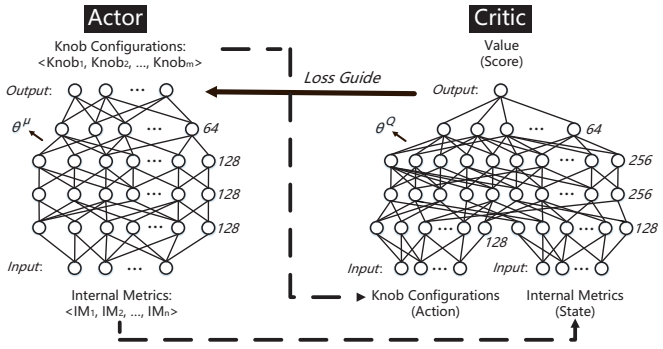


Figure 4: DDPG for CDBTune.

Parameters of critic can be updated with gradient descent. As for actor, we will apply the chain rule and update it with policy gradient derived from $Q(s_t, a_t|\theta^Q)$:

$$\nabla_{\theta^\mu} J \approx \mathbb{E}[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)}]$$
$$= \mathbb{E}[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_t}]$$

The algorithm contains seven main steps (the pseudo code is summarized in Appendix B.1).

Step 1. We first extract a batch of transition (s_t, r_t, a_t, s_{t+1}) from the experience replay memory.

Step 2. We feed s_{t+1} to the actor network and output the knob settings a'_{t+1} to be executed at next moment.

Step 3. We get the value (score) V_{t+1} after sending s_{t+1} and a'_{t+1} to the critic network.

Step 4. According to Q-Learning algorithm, V_{t+1} is multiplied by discount factor γ and added by the value of reward at time t, and now we can estimate the value of V'_t of the current state s_t.

Step 5. We feed s_t (obtained at the first step) to the critic network and further acquire the value V_t of the current state.

Step 6. We compute the square difference between V'_t and V_t and optimize parameter θ^Q of the critic network by gradient descent.

Step 7. We use $Q(s = s_t, \mu(s_t)|\theta^Q)$ outputted by the critic network as the loss function, and adopt gradient descent means to guide the update of the actor network in order that the critic network gives a higher score for the recommendation outputted by the actor network each time.

To make it easier to understand and implement our algorithm, we elaborate the network structure and specific parameters values of DDPG in Table 5 of Appendix B.2. Note that we also discuss the impact of recommended configurations by different networks on the system's performance in Appendix C.2.

Remark. Traditional machine learning methods rely on massive training samples to train the model while we adopt the try-and-error method to make our model generate diversified samples and learn towards an optimizing direction by using deep reinforcement learning, which can reasonably use limited samples to achieve great effects. The main reason why we can achieve great result with limited samples is summarized as follows. Firstly, for solving traditional game problems, we cannot evaluate how the environment will change after taking an action, because the environment of game is random (for example, in Go, we do not know what the opponent will do next). However, in our DBMS tuning, after a configuration is executed, the database (environment) will not randomly change, owing to the dependencies between knobs. Because of this, with relatively fewer samples, it is easier to learn the DBMS tuning model than game problems. Secondly, when modeling our CDBTune, we adopt few input (63 dimensions) and knobs output (266 dimensions), which makes the network efficiently converge without too many samples. Thirdly, RL essentially requires diverse samples, not only massive samples. For example, RL solves the game problem by processing each frame of the game screen to form initial training samples. Each frame time is very short, leading to high training images redundancy. Instead, for DBMS tuning, we will change the parameters of database and collect the performance data. These data are diverse in our learning process. RL can constantly update and optimize the performance with these diverse data. Lastly, coupled with our efficient reward function, our method performs effectively with a small number of diverse samples.

In summary, the DDPG algorithm makes it feasible for deep neural networks to process high-dimensional states and generate continuous actions. DQN is not able to directly map states to continuous actions for maximizing the action-value function. In DDPG, the actor can straightforwardly predict the values for all tunable knobs at the same time without considering the Q-value with specific action and state.

4.2 Reward Function

The reward function is vital in RL, which provides impactful feedback information between the agent and environment. We desire that the reward can simulate the DBA's empirical judgement for real environment in the tuning process.

Next we describe how CDBTune simulates DBA's tuning process to design reward functions. First, we introduce DBA's tuning process as follows:

(1) Suppose that the initial performance of DBMS is D_0 and the final performance tuned by DBA is D_n.

(2) DBA tunes the knobs and the performance becomes D_1 after the first tuning. Then DBA computes the performance change $\Delta(D_1, D_0)$.

(3) At the i-th tuning iteration, DBA expects that the current performance is better than that of the previous one (i.e., D_i is better than D_{i-1} where $i < n$), because DBA aims to improve the performance through the tuning. However, DBA cannot guarantee D_i is better than D_{i-1} at every iteration. To this

end, DBA compares (*a*) D_i and D_0 and (*b*) D_i and D_{i-1}. If D_i is better than D_0, the tuning trend is correct and the reward is positive; otherwise the reward is negative. The reward value is calculated based on $\Delta(D_i, D_0)$ and $\Delta(D_i, D_{i-1})$.

Based on the above idea, we model the tuning method of DBAs, which not only considers the change of performance at the previous time but also the initial time (when the database needs to be tuned). Formally, let r, T and L denote reward, throughput and latency. Especially, T_0 and L_0 are respectively the throughput and latency before tuning. We design the reward function as follows.

At time t, we calculate the rate of performance change Δ from time $t-1$ and the initial time to time t respectively. The detailed formula is shown as follows:

$$\Delta T = \begin{cases} \Delta T_{t\to 0} = & \dfrac{T_t - T_0}{T_0} \\ \Delta T_{t\to t-1} = & \dfrac{T_t - T_{t-1}}{T_{t-1}} \end{cases} \tag{4}$$

$$\Delta L = \begin{cases} \Delta L_{t\to 0} = & \dfrac{-L_t + L_0}{L_0} \\ \Delta L_{t\to t-1} = & \dfrac{-L_t + L_{t-1}}{L_{t-1}} \end{cases} \tag{5}$$

According to Eq. (4) and Eq. (5), we design the reward function below:

$$r = \begin{cases} ((1+\Delta_{t\to 0})^2 - 1)|1+\Delta_{t\to t-1}|, \Delta_{t\to 0} > 0 \\ -((1-\Delta_{t\to 0})^2 - 1)|1-\Delta_{t\to t-1}|, \Delta_{t\to 0} \leqslant 0 \end{cases} \tag{6}$$

Considering that the ultimate goal of tuning is to achieve better performance than the initial settings, we need to reduce the impact of the intermediate process of tuning for designing reward function. Therefore, when the result in Eq. (6) is positive and $\Delta_{t\to t-1}$ is negative, we set the $r = 0$.

According to the above steps we can calculate the rewards of throughput and latency r_T and r_L. Then we multiply these two rewards by different coefficients C_L and C_T, where $C_L + C_T = 1$. Note that the coefficients can be set based on user preferences. We use r to denote the sum of rewards of throughput and latency:

$$r = C_T * r_T + C_L * r_L \tag{7}$$

If the goal of optimization is throughput and latency, our reward function does not need to change, because the reward function is independent to the hardware environment and workload changes, and it only depends on the optimization goal. Thus, the reward function needs to be redesigned only if the optimization goals are changed.

Note other reward functions can be integrated into our system. For verifying the superiority of our designed reward function in the training and tuning process, we compare it with other three typical reward functions in Appendix C.1.1. Moreover, we explore how the two coefficients C_L and C_T will affect the performance of DBMS in Appendix C.1.2.

4.3 Advantages

The advantages of our method are summarized as follows. (1) **Limited Samples.** In the absence of high-quality empirical data, accumulating experience via try-and-error method reduces the difficulty of data acquisition and makes our model generate diverse samples and learn towards an optimizing direction by using deep RL which can reasonably use limited samples to achieve great effects. (2) **High-dimensional Continuous Knobs Recommendation.** The DDPG method can recommend a better configuration recommendation in high-dimensional continuous space than simple regression OtterTune used. (3) **End-to-End Approach.** The end-to-end approach reduces the error caused by multiple segmented tasks and improves the precision of recommended configuration. Moreover, the importance degree of different knobs is treated as an abstract feature which can be perceived and automatically accomplished by deep neural network instead of using an extra method to rank the importance degree of different knobs (see Section 5.2.2). (4) **Reducing the Possibility of Local Optimum.** CDBTune may not find the global optimum, but RL adopts the well-known exploration & exploitation dilemma, which not only gives full play to the ability of the model, but also explores more optimal configurations that the DBA never tried, reducing the possibility of trapping in local optimum. (5) **Good Adaptability.** Different from supervised or unsupervised learning, RL has the ability to simulate the human brain to learn as much as possible in a reasonable direction from experience rather than a specific value, which does not depend on labels or training data too much and adapts to different workloads and hardware configurations in cloud environment with a good adaptability. Note some other ML solutions can be explored to improve the database tuning performance further.

5 EXPERIMENTAL STUDY

In this section, we evaluate the performance of CDBTune and compare with existing approaches. We first show the execution time of our model, then compare with baselines and show the performance of varying tuning steps. Then, we compare the performance of CDBTune with BestConfig [55], OtterTune [4], and 3 DBA experts who have been engaged in tuning and optimizing DBMSs for 12 years in Tencent. Finally, we verify the adaptability of the model under different conditions. We also provide more experiments in Appendix C, including different reward functions, different neural networks and other databases.

Workload. Our experiments are conducted using three kinds of benchmark tools: Sysbench, MySQL-TPCH and TPC-MySQL.

Table 1: Database instances and hardware configuration.

Instance	RAM (GB)	Disk (GB)
CDB-A	8	100
CDB-B	12	100
CDB-C	12	200
CDB-D	16	200
CDB-E	32	300
CDB-X1	(4, 12, 32, 64, 128)	100
CDB-X2	12	(32, 64, 100, 256, 512)

We carry out the experiments with 6 workloads consisting of read-only, write-only and read-write workloads of Sysbench, TPC-H[6] workloads, TPC-C workloads and YCSB workloads, which is similar to existing work. Under Sysbench workloads, we set up 16 tables of which each contains about 200K records (about 8.5GB) and the number of threads is set to 1500. For TPC-C (OLTP), we select the database consisting of 200 warehouses (about 12.8GB) and set the number of concurrent connections to 32. TPC-H (OLAP) workloads contain 16 tables (about 16GB). For YCSB (OLTP), we generate 35GB data using the threads of 50 and operations of 20M. In addition, read-only, write-only and read-write of Sysbench are abbreviated as RO, WO and RW respectively. When we conduct online tuning using the model trained on another condition, the expression is defined as M_{training condition}→{tuning condition}. For example, we use 8GB RAM as a training setting and use it for online tuning on 12GB RAM, and then we denote it as $M\_8G \rightarrow 12G$.

DBA Data. It is hard to collect a large amount of DBA's experience tuning data. In order to reproduce and make a comparison with OtterTune [4], we utilize all the DBA's experience data we accumulated as well as the training data we used on CDBTune to train OtterTune. The proportion of these two kinds of data is about 1:20. For example, considering recommending the configuration for 266 knobs, CDBTune collects 1500 samples, while besides these samples OtterTune will also use 75 historical experience tuning data of the DBA.

Setting. Our CDBTune is implemented using PyTorch[7] and Python tools including scikit-learn library[8]. All the experiments are run on Tencent's cloud server with 12-core 4.0GHz CPU, 64GB RAM and 200GB Disk. We use seven types of CDB's instances in the evaluations and their hardware configurations are shown in Table 1. The difference is mainly reflected in the memory size and disk capacity. For fair comparison, in all experiments, we select the best result of the recommendations of CDBTune and OtterTune in the first 5 steps. Comprehensively, considering that BestConfig (a search-based method) needs to restart the search each time which will take a lot of time, we give 50 steps in the experiment. When the 50 steps are finished, we suspend BestConfig and use the recommended configuration corresponding to its best performance. Last but not least, to improve the offline training performance, we add the method of priority experience replay [38] to accelerate the convergence, which increases the convergence speed by a factor of two (half the number of iterations). We also adopt parallel computing (30 servers) which greatly reduces the offline training time (we do not use parallel computing for online tuning).

[6]http://www.tpc.org/tpch
[7]https://pytorch.org
[8]http://scikit-learn.org/stable

5.1 Efficiency Comparison

We first evaluate the execution time details of our method, then compare with baselines and show the performance of varying tuning steps.

5.1.1 Execution Time Details of CDBTune. In order to know how long a step will take in the training and tuning process, we record the average runtime of each step. The runtime for each step is 5 minutes, which is mainly divided into 5 parts (excluding 2 minutes to restart the CDB) as follows:
(1) **Stress Testing Time (152.88 sec):** Runtime of workload generator for collecting current metrics of the database.
(2) **Metrics Collection Time (0.86 ms):** Runtime of obtaining state vectors from internal metrics and calculating reward from external metrics.
(3) **Model Update Time (28.76 ms):** Runtime of forward computation and back-propagation in the network during one training process.
(4) **Recommendation Time (2.16 ms):** Runtime from inputting the database state to outputting recommended knobs.
(5) **Deployment Time (16.68 sec):** Runtime from outputting recommended knobs to deploying the configurations according to CDB's API interface.

Offline Training. CDBTune takes about 4.7 hours for 266 knobs and 2.3 hours for 65 knobs in offline training. Note that the number of knobs affects the offline training time but will not affect the online tuning time.

Online Tuning. For each tuning request, we run CDBTune in 5 steps, and the online tuning time is 25 minutes.

5.1.2 Comparison with Baselines. We compare the online tuning efficiency of CDBTune with OtterTune [4], BestConfig [55] and DBA. Note that only CDBTune requires offline training. But it trains the model once and uses the model to do online tuning, while OtterTune requires to train the model for every online tuning request and BestConfig requires to do online search. As shown in Table 2, for each tuning request, OtterTune takes 55 minutes, BestConfig takes about 250 minutes, DBAs take 8.6 hours, while CDBTune takes 25 minutes. Note that we invite 3 DBAs to tune the parameters and select the best performance of their results which take 8.6 hours for each tuning request. (We have recorded 57 tuning requesters from 3 DBAs, which totally took 491 hours.) It takes DBA about 2 hours to constantly execute

Table 2: Detailed online tuning steps and time of CDBTune and other tools.

Tuning Tools	Total Steps	Time of One Step (mins)	Total Time (mins)
CDBTune	5	5	25
OtterTune	5	11	55
BestConfig	50	5	250
DBA	1	516	516

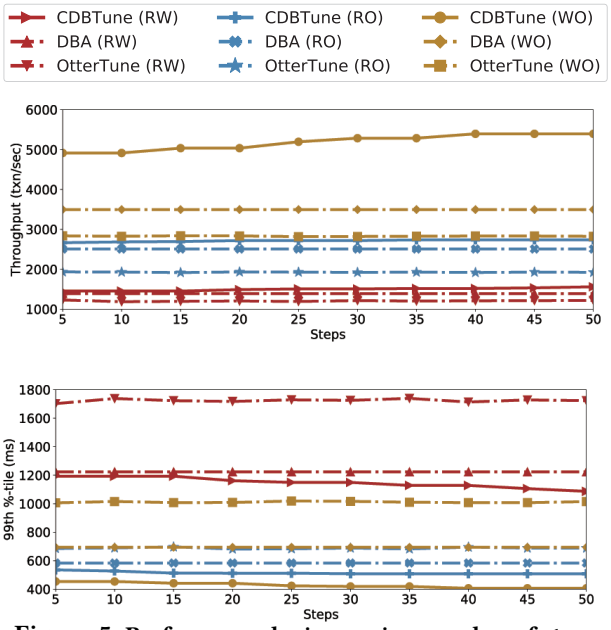


Figure 5: Performance by increasing number of steps

workload replay and detect the factors (e.g., analyzing the most time-consuming functions in the source code, then locating the reason, and finding the corresponding knobs to tune) that affect the performance of DBMS. This process usually requires rich experience and a lot of time cost. OtterTune adopts simple GP regression, and the knobs recommended are not accurate. Therefore, it has to try more trials to achieve a better performance. BestConfig restarts the entire search processing whenever a new tuning request comes, and fails to utilize knowledge gained from previous tuning efforts, thus it requires a lot of trial time.

5.1.3 Varying Tuning Steps. When recommending configurations for the user, we need to replay the current user's workload and conduct stress testing on the instance. In this process, we fine-tune the standard model with limited steps called accumulated trying steps. Hence, how many steps will it take to achieve the desired performance? In most cases, the algorithm will bring better results. However, due to the exploration & exploitation dilemma in DDPG, such knob settings that the past experience never tried will be obtained with a small probability. These outliers may either degrade or improve performance to an unprecedented level. We set 5 steps as a statistical increment unit to observe the system's performance and take the recommended result corresponding to the optimal performance. We carry out experiments

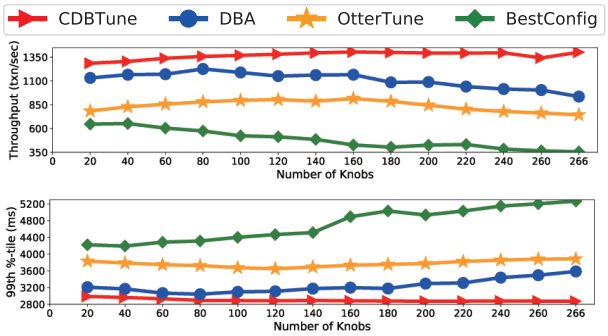


Figure 6: Performance by increasing number of knobs (knobs sorted by DBA).

with CDB-A on three kinds of Sysbench's workloads respectively as shown in Figure 5, where the horizontal coordinate represents the number of tuning steps before recommending configurations while the vertical ordinate represents the value of throughput or latency. It is obvious that the standard model gradually adapts to the current workload through fine-tuning operations as the number of steps increases, which brings continuous improvement to the performance. Also, compared with OtterTune and DBA, CDBTune has already achieved a better result in the first 5 steps in all cases, indicating that our model owns high efficiency, so we believe our algorithm really draws lessons from past experience and performs excellently. Certainly, the user will get better configurations to achieve higher performance if accepting a longer tuning time. However, OtterTune keeps stable with the increasing number of iterations, which is caused by the characteristics of supervised learning and regression.

5.2 Effectiveness Comparison

In this section, we evaluate the effect on different number of knobs and discuss the performance of CDBTune, DBA, OtterTune and Bestconfig with different workloads. With the database instance CDB-B, we record the throughput, latency, and number of iterations for TPC-C workload when the model converges. Note that some knobs do not need to be tuned, e.g., those knobs that do not make sense (e.g., path names) to tune or those that are not allowed to tune (which may lead to hidden or serious problems). Such knobs are added to the black-list according to the DBA or user's demand. We finally sort 266 tunable knobs (the maximum number of knobs that the DBA uses to tune for CDB).

5.2.1 Knobs Selected by DBA and OtterTune. Both DBA and OtterTune rank the knobs based on their importance to the database performance. We use their orders to sort the 266 knobs and select different numbers of knobs following the order to tune and compare different methods. Figures 6 and 7 show the performance with CDB-B under TPC-C workload based on the orders of DBA and OtterTune respectively. We can see from the results that CDBTune can

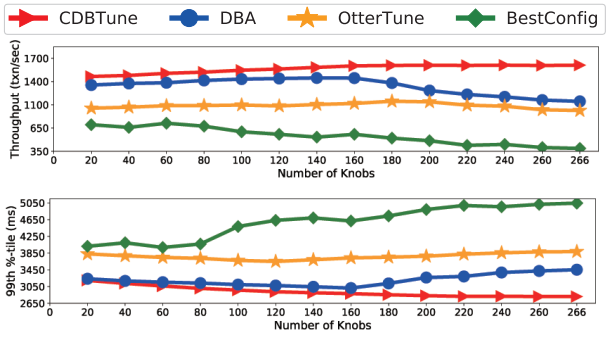


Figure 7: Performance by increasing number of knobs (knobs sorted by OtterTune).

achieve better performance in all cases. Note that the performance of DBA and OtterTune begins to decrease after their recommended knobs exceed a certain number. The main reason is that the unseen dependencies between knobs become more complex in a larger knobs space, but DBA and OtterTune cannot recommend reasonable configurations in such high-dimensional continuous space. This experiment demonstrates that CDBTune is able to recommend better configurations than DBA and OtterTune in high-dimensional continuous space.

5.2.2 Knobs Randomly Selected by CDBTune. CDBTune randomly selects different numbers of knobs (note that the 40 selected knobs must contain the 20 selected knobs from the previous one) and record the performance of CDB-B under TPC-C workload after executing these configurations. As shown in Figure 8, as the number of knobs increases from 20 to 266, the performance of configurations recommended by CDBTune is continuously improved. The performance is poor at the beginning, because a small number of selected knobs have a small impact on performance. Besides, after the number of knobs reach a certain number, the performance tends to be stable. This is also because the added knobs later will not greatly affect the performance. This experiment demonstrates that DBA and OtterTune separately rank the importance of knobs, but our CDBTune automatically complete this process (feature extraction) by deep neural network without additional ranking step (DBA and OtterTune need), which is also in line with our original intention of designing end-to-end model.

In addition, the input and output of the network become larger as the number of knobs increases, resulting that CDBTune takes more steps in the offline training process. Therefore, we use the method of priority experience replay and adopt parallel computing to accelerate the convergence of our model. According to the time cost of each step mentioned in section 5.1, the average time spent on offline training is about 4.7 hours. This time will be further shortened if GPU is used.

5.2.3 Performance Improvement. We also evaluate our method on different workloads with CDB-A and the result

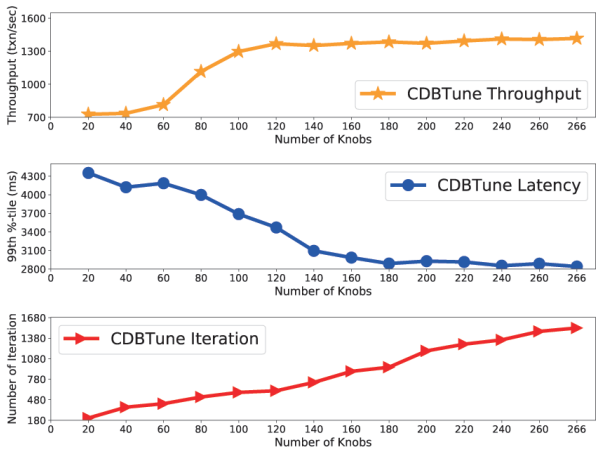


Figure 8: Performance by increasing number of knobs (knobs randomly selected by CDBTune).

Table 3: Higher throughput (T) and lower latency (L) of CDBTune than BestConfig, DBA and OtterTune.

Workload	BestConfig		DBA		OtterTune	
	T	L	T	L	T	L
RW	↑ 68.28%	↓ 51.65%	↑ 4.48%	↓ 8.91%	↑ 29.80%	↓ 35.51%
RO	↑ 42.15%	↓ 43.95%	↑ 4.73%	↓ 11.66%	↑44.46%	↓ 23.63%
WO	↑ 128.66%	↓ 61.35%	↑ 46.57%	↓ 43.33%	↑ 91.25%	↓ 59.27%

is shown in Figure 9. The tuning performance improvement percentage is shown in Table 3 compared with BestConfig, DBA and OtterTune. It can been seen that CDBTune achieves higher performance than OtterTune, which in turn is better than BestConfig. Consequently, the learning-based method is more effective and our algorithm obtains the state-of-the-art result. Besides, OtterTune performs inferior to the DBA in most cases. This is because we use the try-and-error samples in RL instead of massive high-quality DBA's experience tuning data. Compared with BestConfig, we find that CDBTune greatly outperforms it, because in a short time, BestConfig can hardly find the optimal configurations without any past experience in a high-dimensional space. This verifies that the learning-based approach has overwhelming predominance in achieving better solution quickly than search-based tuning, and also verifies the superiority of CDBTune.

CDBTune is able to achieve better performance than other candidates, especially gains a remarkable improvement under the write-only workload. We observe except that the buffer pool size is enlarged, configurations which CDBTune recommended also expand the size of log file properly. In addition, *innodb_read_io_threads* will increase under the RO workload while both *innodb_write_io_threads* and *innodb_purge_threads* are becoming appropriately larger when the workload is WO or RW. This shows that our model can properly tune knobs under different workloads, which improve the CPU utilization as well as the database performance. Of course, for those knobs like *innodb_file_per_table*,

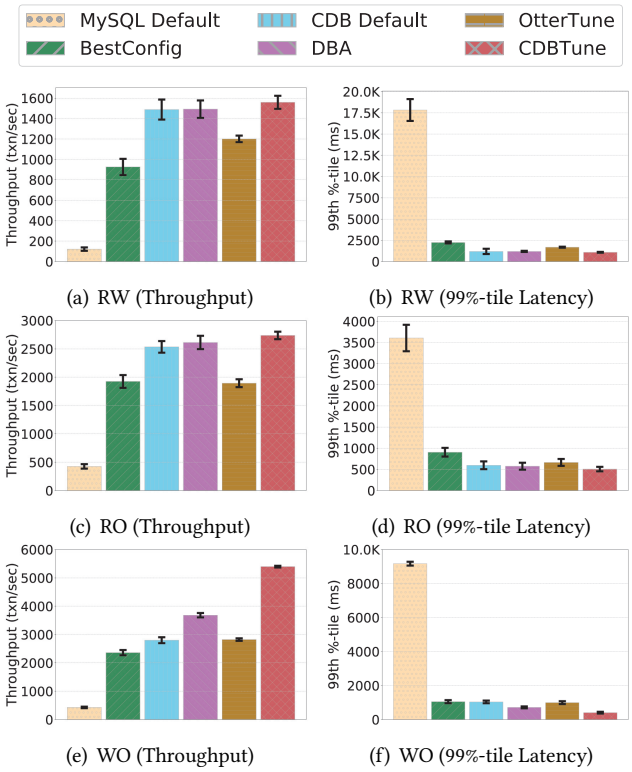


Figure 9: Performance comparison for Sysbench RW, RO and WO workload among CDBTune, MySQL default, BestConfig, CDB default, DBA and OtterTune.

max_binlog_size and *skip_name_resolve*, we obtain the same values as the DBA advisors. According to the MySQL official manual regulations, the product of *innodb_log_files_in_group* and *innodb_log_file_size* is not allowed to be greater than the value of disk capacity. Also, we find that during the real training process of our model, the CDB's instance will easily crash once the product exceeds the threshold, because the log files take up too much disk space, resulting that more data cannot be written. An interesting finding is that faced with this situation, we do not limit the range of these two parameters but give a large negative reward (e.g., -100) for punishment. Instead, the practical results verify this method achieves a good effect with the constant reward feedback in RL and this situation is becoming less and less, and even disappears as the training process goes on, although the crash may frequently occur in the initial training. Ultimately, the product of the two values recommended by our model is reasonable which will not result in the crash.

5.3 Adaptability

We evaluate the adaptability, e.g., how a method can adapt to new environment or new workload.

5.3.1 Adaptability on Memory Size and Disk Capacity Change. Compared with local self-built databases, one of

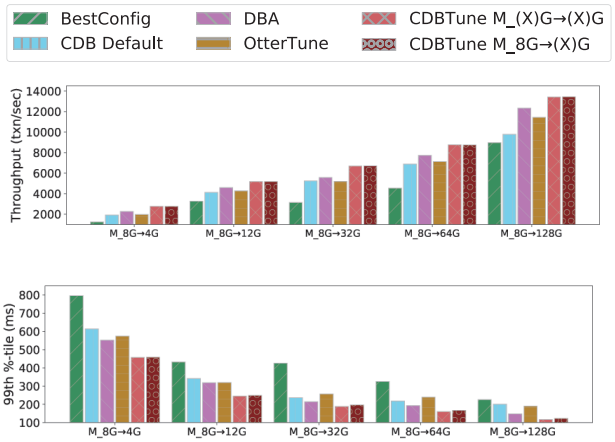


Figure 10: Performance comparison for Sysbench WO workload when applying the model trained on 8G memory to (X)G memory hardware environment.

the biggest advantages for cloud databases is that data migration or even downtime reloading is hardly required when resources need to be adjusted. Usually, memory size and disk capacity are the most two properties that users prefer to adjust. Thus, in the cloud environment, so many different users own their respective cloud database memory size and disk capacity that we are not able to build a corresponding model for each one. Therefore, this cloud environment naturally requires the DBMS tuning models own good adaptability. In order to verify that CDBTune can greatly optimize the database's performance with different hardware configurations, we use database instance CDB-A (8G RAM, 100G Disk), CDB-X1 (XG RAM, 100G Disk) where X is selected from (4, 12, 32, 64, 128), CDB-C (12G RAM, 200G Disk) and CDB-X2 (12G RAM, XG Disk) where X is selected from (32, 64, 100, 256, 512). Note that there is only a different memory size between CDB-A and CDB-X1 while just a disk capacity difference between CDB-C and CDB-X2. For different memory size, under write-only workloads, we first directly utilize the model called M_A→X1 trained on CDB-A to recommend configurations for CDB-X1 (cross testing), then use the model called M_X1→X1 trained on CDB-X1 to recommend configurations for CDB-X1 (normal testing), and finally compare the performance after executing these two configurations. Similarly, for different disk capacity, we utilize the same way to complete cross testing and normal testing on CDB-C and CDB-X2. As shown in Figure 10 and Figure 11, the cross-testing model almost achieves the same performance as normal-testing model. Moreover, both of above two models achieve better performance than OtterTune, BestConfig and the DBAs employed by Tencent's cloud database, indicating that our CDBTune does not need to establish a new model and owns a strong adaptability that can completely adapt to a new hardware environment no matter how memory size, disk capacity of users change.

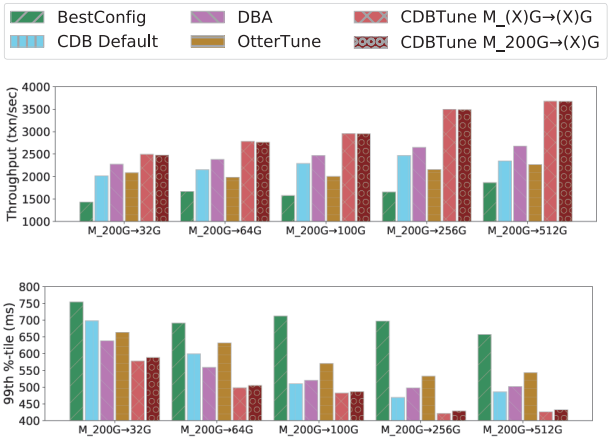


Figure 11: Performance comparison for Sysbench RO workload when applying the model trained on 200G disk to (X)G disk hardware environment.

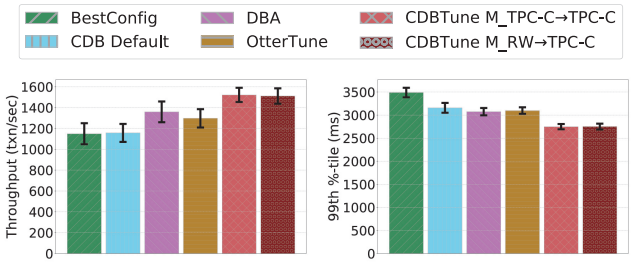


Figure 12: Performance comparison when applying the model trained on Sysbench RW workloads to TPC-C.

5.3.2 Adaptability on Workload Change. As mentioned above, we adopt some standard testing tools to generate samples data for training in the absence of enough experience data. By this means, can the trained CDBTune adapt to other workloads? With database instance CDB-C, we utilize the model called M_TPC-C→TPC-C trained on TPC-C workload to recommend configurations for TPC-C workload (normal testing) as well as use the model called M_RW→TPC-C trained on the read-write workload contained in Sysbench to recommend configurations for TPC-C workload (cross testing). After deploying these two configurations recommended by CDBTune on CDB, we record their respective performance in the last two bars as shown in Figure 12. Also, the tuning performance of cross-testing model is slightly different from that of normal-testing model. That indicates that our CDBTune does not need to establish a new model and owns a good adaptability when the workload changes slightly.

Summary. The results of above three experiments show that our model with limited training data manifests a strong adaptability no matter how the environment or data changes. The fundamental reason is that OtterTune relies highly on training datasets, and uses simple regression for recommendation like GP regression. Thus the performance of recommended configuration is very limited, and OtterTune and BestConfig do not search high-dimensional configuration space when the current workload or hardware configuration differs from training condition. Especially in cloud environment, when the external environment changes, lacking relevant data in the training dataset will directly bring a poor recommendation to OtterTune. Instead, RL makes CDBTune simulate human brain, learn towards an optimizing direction, and recommend reasonable knob settings corresponding to the current workload and hardware environment. Thus, CDBTune owns a good adaptability in cloud environment. In addition, we have conducted similar experiments on different hardware media, e.g., SSD and NVM, and we get similar results, which are omitted due to the limited space.

6 RELATED WORK

Database Tuning. DBMS tuning has been an interesting and active area of research in the last two decades [1, 4, 6, 7, 10, 11, 16, 22, 37, 51, 52, 56, 57]. Existing work can be classified into two broad categories: tuning the physical design and tuning the configuration parameters.

(1) Physical Design Tuning. Major database vendors offer tools for automating database physical design [6, 9, 53, 57], and they focused on index optimizations, materialized views and partitions [1–3, 8, 23, 28, 36]. Database cracking is a technique to create indexes adaptively and incrementally as a side-product of query processing [19]. Several studies have proposed different cracking techniques for different aspects including tuple reconstruction [20], updates [16], and convergence [21]. Schuhknecht et al. conducted an experimental study on database cracking to identify the potential, and proposed promising directions in database cracking [39]. Richter et al. presented a novel indexing approach for HDFS and Hadoop MapReduce to create different clustered indexes over terabytes of data with minimal costs [37]. Idreos et al. presented the Data Calculator to enable interactive and semi-automated design of data structures and performance by capturing the first principles of data layout design and using learned cost models, respectively [22].

(2) Database Configuration Tuning. A parameter configuration tuning selects appropriate values of parameters (knobs) that can improve DBMS's performance. Most work in automated database tuning has either focused on specific parameter tuning (e.g., [41]) or holistic parameter tuning (e.g., [12]).

(i) Specific Parameter Tuning. Techniques for tuning specific classes of parameters include memory managements and identifying performance bottlenecks [19, 21, 39]. IBM DB2 released a self-tuning memory manager that uses heuristics to allocate memory to the DBMS's internal components [41, 45]. Tran et al. used linear and quadratic regression models for buffer tuning [46]. A resource monitoring tool has been used with Microsoft's SQL Server for the self-predicting

DBMS [33]. Oracle also developed an internal monitoring system to identify bottlenecks due to misconfigurations [13, 25]. The DBSherlock tool helps a DBA diagnose problems by comparing slow regions with normal regions based on the DBMS's time-series performance data [54].

(ii) Holistic Parameter Tuning. There are several works for holistic tuning of many configuration parameters in modern database systems. The COMFORT tool uses a technique from control theory that can adjust a single knob up or down at a time, but cannot discover the dependencies between multiple knobs [50]. IBM DB2 released a performance Wizard tool for automatically selecting the initial values for the configuration parameters [26]. BerkeleyDB uses influence diagrams to model probabilistic dependencies between configuration knobs, to infer expected outcomes of a particular DBMS configuration [42]. However, these diagrams must be created manually by a domain expert. The SARD tool generates a relative ranking of a DBMS's knobs using the Plackett-Burman design [12]. iTuned is a generic tool that continuously makes minor changes to the DBMS configuration, employing the GP regression for automatic configuration tuning [14].

Our system is designed for holistic knob tuning. OtterTune [4] is most close to our work. It is a multistep tuning tool to select the most impactful knobs, map unseen database workloads to previous workloads and recommend knob settings. However, the dependencies between each step make the whole process relatively complicated. And OtterTune requires a lot of high-quality samples which are hard to collect in cloud environment. BestConfig [55] is the closest work that is related to our goals but the techniques are completely different. It divides the high-dimensional parameter space into subspaces, and exploits the search-based methods. However, it does not learn experience from previous tuning efforts (i.e., even if there are two identical cases, it will search twice). CDBTune is an end-to-end tuning system, only requiring a few samples to tune cloud database. The experimental results show that our method achieves much better performance than OtterTune and BestConfig.

Deep Learning for Database. Deep learning models define a mapping from an input to an output, and learn how to use the hidden layers to produce correct output [27]. Although deep learning has successfully been applied to solving computationally intensive learning tasks in many domains [15, 17, 18, 24, 48], there are few studies that have used deep learning techniques to solve database tuning problems[49]. Reinforcement learning is able to discover the best action through the try-and-error method by either exploiting current knowledge or exploring unknown states to maximize a cumulative reward [30, 43, 44].

Recently, several researches utilized deep learning or RL model to solve the database problems. Tzoumas et al. [47] transformed the query plans building into an RL problem where each state represents a tuple along with metadata about which operators still need to be applied and each action represents which operator to run next. Basu et al. [5] used RL model to adaptive performance tuning of database applications. Pavlo et al. [35] presented the architecture of Peloton for workload forecasting and action deployment under the algorithmic advancements in deep learning. Marcus et al. [31] used deep RL to determine join order. Ortiz et al. [34] used deep RL to incrementally learn state representations of subqueries for query optimization. It models each state as a latent vector that is learned through a neural network and is propagated to other subsequent states. Sharma et al. [40] used the deep RL model to automatically administer a DBMS by defining a problem environment.

Our tuning system uses a deep reinforcement learning model for automatic DBMS tuning. The goal of CDBTune is to tune the knob settings for improving the performance of cloud databases. To the best of our knowledge, this is the first attempt that uses deep RL model for configuration recommendation in databases.

7 CONCLUSION

In this paper, we proposed an end-to-end automatic DBMS configuration tuning system CDBTune that can recommend superior knob settings in the complex cloud environment. CDBTune used a try-and-error manner in RL to learn the best settings with limited samples. Besides, our designed reward function can effectively improve tuning efficiency and the DDPG algorithm can find the optimal configurations in high-dimensional continuous space. Extensive experimental results showed that CDBTune produced superior configurations for various workloads that greatly improved performance with higher throughput and lower latency compared with the state-of-the-art tuning tools and DBA experts. We also demonstrated CDBTune had a good adaptability whenever the operating environment changed. Note that some other ML solutions can be explored to improve the database tuning performance further.

ACKNOWLEDGMENTS

Thanks for the research fund of the Intelligent Cloud Storage Joint Research center of HUST and Tencent, the Key Laboratory of Information Storage System, Ministry of Education. This work is supported by the Innovation Group Project of the National Natural Science Foundation of China, No.61821003, the National Natural Science Foundation of China No.(61632016, 61472198), 973 Program of China No.2015CB358700 and the National Key Research and Development Program of China under Grant No.2016YF-B0800402.

REFERENCES

[1] Sanjay Agrawal, Nicolas Bruno, Surajit Chaudhuri, and Vivek R Narasayya. 2006. AutoAdmin: Self-Tuning Database SystemsTechnology. *IEEE Data Eng. Bull.* 29, 3 (2006), 7–15.

[2] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. 2005. Database tuning advisor for microsoft sql server 2005. In *ACM SIGMOD*. ACM, 930–932.

[3] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. 2004. Integrating vertical and horizontal partitioning into automated physical database design. In *ACM SIGMOD*. ACM, 359–370.

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

[5] Debabrota Basu, Qian Lin, Hoang Tam Vo, Hoang Tam Vo, Zihong Yuan, and Pierre Senellart. 2016. *Regularized Cost-Model Oblivious Database Tuning with Reinforcement Learning*. Springer Berlin Heidelberg. 96–132 pages.

[6] Peter Belknap, Benoit Dageville, Karl Dias, and Khaled Yagoub. 2009. Self-tuning for SQL performance in Oracle database 11g. In *ICDE*. IEEE, 1694–1700.

[7] Phil Bernstein et al. 1998. The Asilomar report on database research. *ACM Sigmod record* 27, 4 (1998), 74–80.

[8] Nicolas Bruno and Surajit Chaudhuri. 2005. Automatic physical database tuning: a relaxation-based approach. In *ACM SIGMOD*. ACM, 227–238.

[9] Surajit Chaudhuri and Vivek Narasayya. 1998. AutoAdmin "what-if" index analysis utility. In *ACM SIGMOD*. 367–378.

[10] Surajit Chaudhuri and Vivek Narasayya. 2007. Self-Tuning Database Systems: A Decade of Progress.. In *VLDB*. 3–14.

[11] Surajit Chaudhuri and Gerhard Weikum. 2000. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System.. In *VLDB*. 1–10.

[12] Biplob K Debnath, David J Lilja, and Mohamed F Mokbel. 2008. SARD: A statistical approach for ranking database tuning parameters. In *ICDEW*. IEEE, 11–18.

[13] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. 2005. Automatic Performance Diagnosis and Tuning in Oracle.. In *CIDR*. 84–94.

[14] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with iTuned. *VLDB Endowment* 2, 1 (2009), 1246–1257.

[15] Yoav Goldberg. 2015. A Primer on Neural Network Models for Natural Language Processing. *Computer Science* (2015).

[16] Goetz Graefe and Harumi A. Kuno. 2010. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*. 371–381.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. 770–778.

[18] G. E. Hinton and R. R. Salakhutdinov. 2006. Reducing the dimensionality of data with neural networks. *Science* 313, 5786 (2006), 504–507.

[19] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR*. 68–78.

[20] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2009. Self-organizing tuple reconstruction in column-stores. In *ACM SIGMOD*. 297–308.

[21] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. 2011. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB* 4, 9 (2011), 585–597.

[22] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. 2018. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *ACM SIGMOD*. 535–550.

[23] Ihab F Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. 2004. CORDS: automatic discovery of correlations and soft functional dependencies. In *ACM SIGMOD*. ACM, 647–658.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *NIPS*. 1097–1105.

[25] Sushil Kumar. 2003. Oracle database 10g: The self-managing database.

[26] Eva Kwan, Sam Lightstone, Adam Storm, and Leanne Wu. 2002. Automatic configuration for IBM DB2 universal database. In *Proc. of IBM Perf Technical Report*.

[27] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436.

[28] Sam S Lightstone and Bishwaranjan Bhattacharjee. 2004. Automated design of multidimensional clustering tables for relational databases. In *VLDB*. VLDB, 1170–1181.

[29] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *CoRR* abs/1509.02971 (2015).

[30] Vasilis Maglogiannis, Dries Naudts, Adnan Shahid, and Ingrid Moerman. 2018. A Q-Learning Scheme for Fair Coexistence Between LTE and Wi-Fi in Unlicensed Spectrum. *IEEE Access* 6 (2018), 27278–27293.

[31] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. *arXiv preprint arXiv:1803.00055* (2018).

[32] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR* abs/1312.5602 (2013).

[33] Dushyanth Narayanan, Eno Thereska, and Anastassia Ailamaki. 2005. Continuous resource monitoring for self-predicting DBMS. In *null*. IEEE, 239–248.

[34] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2018. Learning State Representations for Query Optimization with Deep Reinforcement Learning. *arXiv preprint arXiv:1803.08604* (2018).

[35] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems.. In *CIDR*.

[36] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. 2002. Automating physical database design in a parallel database. In *ACM SIGMOD*. ACM, 558–569.

[37] Stefan Richter, Jorge-Arnulfo Quiané-Ruiz, Stefan Schuh, and Jens Dittrich. 2014. Towards zero-overhead static and adaptive indexing in Hadoop. *VLDB J.* 23, 3 (2014), 469–494.

[38] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2015. Prioritized Experience Replay. *Computer Science* (2015).

[39] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. 2013. The Uncracked Pieces in Database Cracking. *PVLDB* 7, 2 (2013), 97–108.

[40] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. The Case for Automatic Database Administration using Deep Reinforcement Learning. (2018).

[41] Adam J Storm, Christian Garcia-Arellano, Sam S Lightstone, Yixin Diao, and Maheswaran Surendra. 2006. Adaptive self-tuning memory in DB2. In *VLDB*. VLDB, 1081–1092.

[42] David G Sullivan, Margo I Seltzer, and Avi Pfeffer. 2004. *Using probabilistic reasoning to automate software tuning*. Vol. 32. ACM.

[43] R S Sutton and A G Barto. 2005. Reinforcement Learning: An Introduction, Bradford Book. *IEEE Transactions on Neural Networks* 16, 1 (2005), 285–286.

[44] Richard S Sutton and Andrew G Barto. 2011. Reinforcement learning: An introduction. (2011).

[45] Wenhu Tian, Pat Martin, and Wendy Powley. 2003. Techniques for automatically sizing multiple buffer pools in DB2. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 294–302.
[46] Dinh Nguyen Tran, Phung Chinh Huynh, Yong C Tay, and Anthony KH Tung. 2008. A new approach to dynamic self-tuning of database buffers. *TOS* 4, 1 (2008), 3.
[47] Kostas Tzoumas, Timos Sellis, and Christian S Jensen. 2008. A reinforcement learning approach for adaptive query processing. *History* (2008).
[48] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. SuperNeurons: Dynamic GPU Memory Management for Training Deep Neural Networks. (2018).
[49] Wei Wang, Meihui Zhang, Gang Chen, HV Jagadish, Beng Chin Ooi, and Kian-Lee Tan. 2016. Database meets deep learning: Challenges and opportunities. *ACM SIGMOD Record* 45, 2 (2016), 17–22.
[50] Gerhard Weikum, Christof Hasse, Axel Mönkeberg, and Peter Zabback. 1994. The COMFORT automatic tuning project. *Information systems* 19, 5 (1994), 381–432.
[51] Gerhard Weikum, Axel Moenkeberg, Christof Hasse, and Peter Zabback. 2002. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *VLDB*. Elsevier, 20–31.
[52] David Wiese, Gennadi Rabinovitch, Michael Reichert, and Stephan Arenswald. 2008. Autonomic tuning expert: a framework for best-practice oriented autonomic database tuning. In *2008 conference of the center for advanced studies on collaborative research: meeting of minds*. ACM, 3.
[53] Khaled Yagoub, Peter Belknap, Benoit Dageville, Karl Dias, Shantanu Joshi, and Hailing Yu. 2008. Oracle's SQL Performance Analyzer. *IEEE Data Eng. Bull.* 31, 1 (2008), 51–58.
[54] Dong Young Yoon, Ning Niu, and Barzan Mozafari. 2016. Dbsherlock: A performance diagnostic tool for transactional databases. In *ACM SIGMOD*. ACM, 1599–1614.
[55] 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.
[56] Daniel C Zilio. 1998. Physical Database Design Decision Algorithms and Concurrent Reorganization for Parallel Database Systems.
[57] Daniel C Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. 2004. DB2 design advisor: integrated automatic physical database design. In *VLDB*. VLDB, 1087–1097.

APPENDIX

A NOTATIONS

All the notations used in the paper are listed in Table 4.

B DETAILS OF RL IN CDBTUNE

B.1 Algorithm of DDPG

The Algorithm of DDPG is shown in Algorithm 1.

B.2 Specific parameters of DDPG

In order to make it easier for readers to understand clearly and implement our algorithm, we elaborate the network structure and specific parameters values of DDPG in Table 5.

Table 4: Notations

Variables	Descriptions	Mapping to CDBTune
s	State	Internal metrics of DBMS
a	Action	Tunable knobs of DBMS
r	Reward	The performance of DBMS
α	Learning rate	Set to 0.001
γ	Discount factor	Set to 0.99
ω	The weights of neural network	Initialized to $Uniform(-0.1, 0.1)$
E	Environment, the tuning target	An instance of CDB
μ	Policy	deep neural network
θ^Q	Learnable parameters	Initialized to $Normal(0, 0.01)$
θ^μ	Actor, mapping state s_t to action a_t	-
Q^μ	Critic, the policy μ	-
L	Loss function	-
y	Q value label through Q-learning algorithm	-

Algorithm 1 Deep deterministic policy gradient (DDPG)

1: Sample a transition (s_t, r_t, a_t, s_{t+1}) from Experience Replay Memory.
2: Calculate the action for state s_{t+1}: $a'_{t+1} = \mu(s_{t+1})$.
3: Calculate the value for state s_{t+1} and a'_{t+1}: $V_{t+1} = Q(s_{t+1}, a'_{t+1}|\theta^Q)$.
4: Apply Q-learning and obtain the estimated value for state s_t: $V'_t = \gamma V_{t+1} + r_t$.
5: Calculate the value for state s_t directly: $V_t = Q(s_t, a_t|\theta^Q)$.
6: Update the critic network by gradient descent and define the loss as:

$$L_t = (V_t - V'_t)^2$$

7: Update the actor network by policy gradient:

$$\nabla_a Q(s_t, a|\theta^Q)|_{a=\mu(s_t)} \nabla_{\theta^\mu} \mu(s_t|\theta^\mu)$$

Table 5: Detailed Actor-Critic network and parameters of DDPG.

Step	Actor		Critic	
	Layer	Param	Layer	Param
1	Input	63	Input	#Knobs + 63
2	Full Connection	128	Parallel Full Connection	128 + 128
3	ReLU	0.2	Full Connection	256
4	BatchNorm	16	ReLU	0.2
5	Full Connection	128	BatchNorm	16
6	Tanh	-	Full Connection	256
7	Dropout	0.3	Full Connection	64
8	Full Connection	128	Tanh	-
9	Tanh	-	Dropout	0.3
10	Full Connection	64	BatchNorm	16
11	Output	#Knobs	Output	1

B.3 Difference between Q-Learning and DQN

The difference between Q-Learning and DQN is shown in Figure 13.

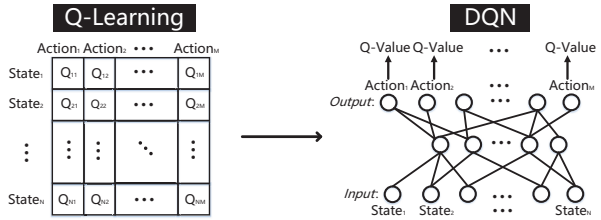


Figure 13: Difference between Q-Learning and DQN.

C MORE EXPERIMENTS

C.1 Evaluation on Reward Functions

C.1.1 Different Reward Functions. For verifying the superiority of our designed reward function in the training and tuning process, we compare it with other three typical reward functions including (1) RF-A: the performance of current time is compared only with that of the previous time, (2) RF-B: the performance of current time is compared only with that of the initial settings, and (3) RF-C: if the current performance is lower compared with that of the previous time, the corresponding reward part will keep the original method of calculation (for example, its reward remains unchanged even if $\Delta_{t\rightarrow t-1}$ is negative in Eq. (6)). We make a comparison between the three reward functions and our designed RF-CDBTune in Section 4.2 in terms of the number of iterations when the model converged in the training process. After multiple steps, if the performance change between two steps does not exceed 0.5% in five consecutive steps, we decide that the model has converged. It is a tradeoff to select an appropriate threshold between training time and model quality. For example, a smaller threshold may achieve a better result but it will spend more time on model training. We have conducted extensive experiments, and found if we set the convergence threshold to 0.5%, our model can converge fast and achieve good results. Due to the limitation of the length of the paper, we do not provide detailed discussion and the performance deployed on the recommended configurations. Specially, compared with the performance of the previous time and initial settings, the reward (corresponding to throughput or latency) calculate by RF-CDBTune will be set to 0 if the current performance is lower than that of the previous time but higher than the initial performance.

As shown in Figure 14, we adopt three different workloads as well as instances CDB-A (8G RAM, 100G Disk) and CDB-C (12G RAM, 200G Disk). As a whole, RF-A takes the longest convergence time. What causes this phenomenon is that RF-A just considers the performance of the previous time, neglecting the final goal that we expect to achieve higher performance than the initial settings as much as possible. Therefore, there is a high chance that a positive reward will be given when the current performance is worse than the initial settings but better than that of the previous time, bringing long convergence time and low performance to the model. RF-B only achieves a sample target which obtains a better result than the initial settings regardless of the previous performance although it takes the shortest convergence time. Instead, RF-B gets the worst performance because it pays no attention to improving the intermediate process. RF-C achieves almost the same performance as RF-A, but spends much more convergence time than RF-A. If the current performance is lower than that of the previous time, the absolute value part of its reward function is always positive but generally a small number, which will produce a small impact on the system's performance. However, such reward misleads the learning of the intermediate process, leading to a longer convergence time than RF-A. In conclusion, compared with others, our proposed RF-CDBTune takes above factors into consideration comprehensively and achieves the fast convergence speed and best performance.

C.1.2 Varying C_T and C_L. In this section, we describe our thought about designing reward function in detail. In Eq. 7, we present two coefficients (C_T and C_L) to separately optimize throughput and latency, where C_T+C_L =1. For exploring how these coefficients will affect the performance of DBMS, we take this performance as a benchmark when setting C_T=C_L=0.5. We change the size of C_T and observe the ratio of throughput to latency compared with that of predefined benchmark. As shown in Figure 15, the throughput increases gradually with a larger C_T. Besides, observing the slope of curve, when C_T exceeds 0.5, we find the change rate of throughput is larger than that of a smaller C_T (less than 0.5). And so is the latency. This is because the change of C_T and C_L will affect the contribution of throughput and latency to reward. For example, a larger C_T can reduce the sensitivity of CDBTune to latency. In general, we set C_T=C_L=0.5. But we also allow different weights on latency and throughput (we set C_L = 0.6 and C_T = 0.4 in our experiments), the user can set the weights and obtain different results according to his own requirement (latency or throughput sensitivity).

C.2 Evaluation by Varying Neural Network

In this section, we discuss the impact of recommended configurations by different networks on the system's performance when tuning 266 knobs. We mainly change the number of hidden layers and neurons in each layer of both Actor and Critic network. The detailed parameters are displayed in Table 6. The number of iterations multiply increases with an increasing number of hidden layers, and the performance will even decrease when the number of layers increases to

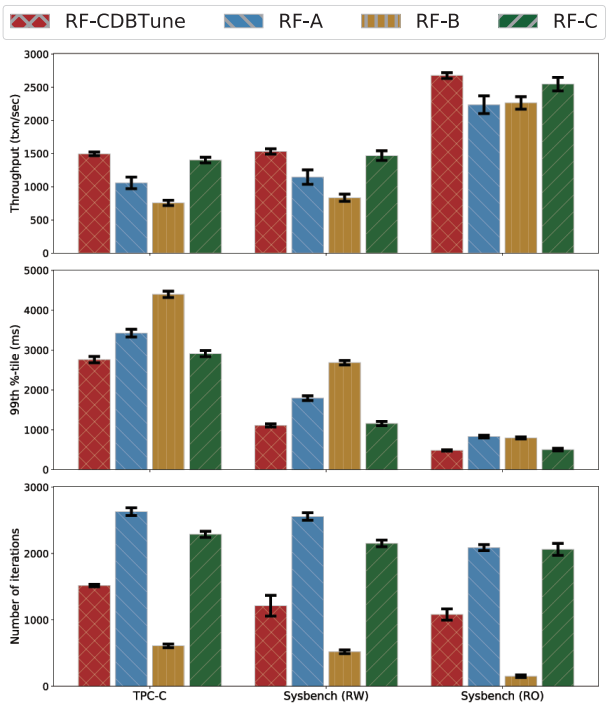


Figure 14: Number of iterations and performance of CDBTune for TPC-C (with CDB-C), Sysbench RW and RO workloads (with CDB-A) respectively using different reward functions. The corresponding number of iterations and performance are collected under the same knob settings.

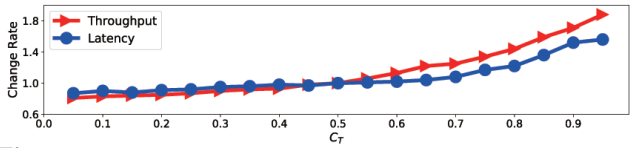


Figure 15: The coefficient C_T to optimize throughput and latency. Note that $C_T + C_L = 1$.

5. This may result from the high complexity of our model due to the increasing number of layers, which leads to overfitting. Moreover, when the number of hidden layers (of both two networks) is fixed, increasing the number of neurons in each layer seems to produce little effect on the performance, but the number of iterations increases a lot due to the higher complexity of the network. Therefore, it is also vital to choose a reasonable and efficient network after fixing the number of knobs, which is why we use the network structure of Figure 4 in Section 4.1.

C.3 Evaluation on Other Databases

We evaluate our method on Local MySQL, MongoDB and Postgres where we tune 232 knobs for MongoDB and 169 knobs for Postgres. Figures 16-17 and Figures 18 show the results. Our method can also work well on Local MySQL, MongoDB and Postgre. For example, our method can adapt

Table 6: Tuning performance varying neural network structure. AHL and CHL are short for the hidden layer in Actor and Critic respectively. The unit of Thr(Throughput) is txn/sec and Lat(Latency) is ms.

AHL	Neurons	CHL	Neurons	Thr	Lat	Iteration
3	128-128-64	3	256-256-64	↓1169.37	↑3042.75	↓682
3	256-256-128	3	512-512-128	↓1195.19	↑3087.58	↓1034
4	**128-128-128-64**	**4**	**256-256-256-64**	**1416.71**	**2840.41**	**1530**
4	256-256-256-128	4	512-512-512-128	↓1394.65	↓2836.27	↑2436
5	128-128-128-128-64	5	256-256-256-256-64	↓1389.47	↑2795.87	↑1946
5	256-256-256-256-128	5	512-512-512-512-128	↓1402.55	↑2801.12	↑3175
6	128-128-128-128-128-64	6	256-256-256-256-256-64	↓1255.78	↑2932.42	↑2564
6	256-256-256-256-256-128	6	512-512-512-512-512-128	↓1305.96	↑2976.53	↑3866

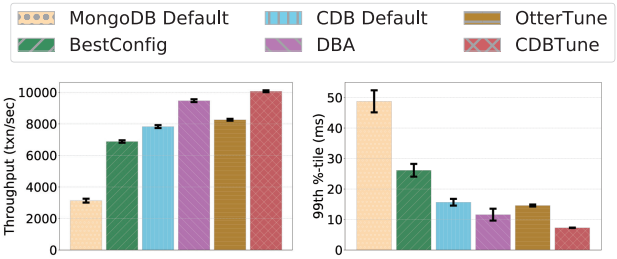


Figure 16: Performance comparison for YCSB workload using instance CDB-E among CDBTune, MongoDB default, CDB default, BestConfig, DBA and OtterTune (on MongoDB).

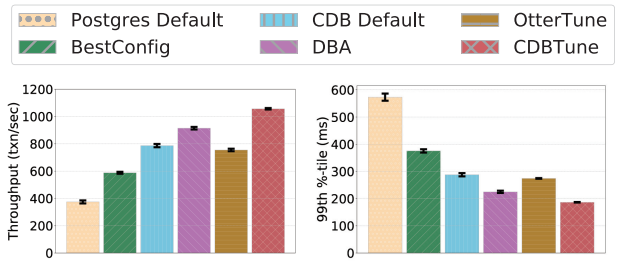
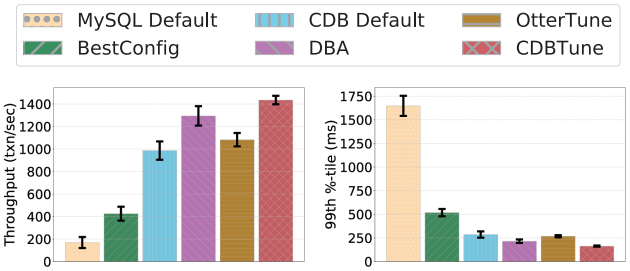


Figure 17: Performance comparison for TPC-C workload using instance CDB-D among CDBTune, Postgres default, CDB default, BestConfig, DBA and OtterTune (on Postgres).



(a) TCP-C (Throughput) (b) TCP-C (99%-tile Latency)

Figure 18: Performance on TPC-C for local MySQL.

to YCSB workloads using the trained model on the CDB-E database instance on MongoDB, TPC-C workloads using the trained model on the CDB-D database instance on Postgres and CDB-C database instance on Local MySQL. CDBTune still achieves the best results and outperforms the state-of-the-art tuning tools and DBA experts significantly. This illustrates that our model is designed with strong scalability in database tuning.