The VLDB Journal (2021) 30:959–987
https://doi.org/10.1007/s00778-021-00670-9

REGULAR PAPER

CDBTune$^+$: An efficient deep reinforcement learning-based automatic cloud database tuning system

Ji Zhang[1,2] · Ke Zhou[1] · Guoliang Li[3] · Yu Liu[1] · Ming Xie[4] · Bin Cheng[4] · Jiashu Xing[4]

Received: 30 January 2020 / Revised: 10 February 2021 / Accepted: 10 April 2021 / Published online: 5 June 2021

Abstract
Configuration tuning is vital to optimize the performance of a database management system (DBMS). It becomes more tedious and urgent for cloud databases (CDB) due to diverse database instances and query workloads, which make the job of a database administrator (DBA) very difficult. Existing solutions for automatic DBMS configuration tuning 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, existing approaches cannot recommend reasonable configurations for a large number of knobs to tune whose potential values live in such high-dimensional continuous space. Lastly, in cloud environments, existing approaches 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 a high-dimensional continuous space. CDBTune$^+$ adopts a trial-and-error strategy to learn knob settings with a limited number of samples to accomplish the initial training, which alleviates the necessity of collecting a massive amount of 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 the efficiency of online tuning. Besides, we propose effective techniques to improve the training and tuning efficiency of CDBTune$^+$ for practical usage in a cloud environment. We conducted extensive experiments under 7 different workloads on real cloud databases to evaluate CDBTune$^+$. Experimental results showed that CDBTune$^+$ adapts well to a new hardware environment or workload, and significantly outperformed the state-of-the-art tuning tools and DBA experts.

Keywords Cloud database · Tuning · Reinforcement learning · Automatic

1 Introduction

The performance of database management systems (DBMSs) relies on hundreds of tunable configuration knobs. We list 65 commonly used knobs for users to tune the performance of their cloud database in Appendix C. For example, the tunable knob *innodb_buffer_pool_size* is the memory space in which indexes, caches, buffers, etc. are stored which specifies the amount of memory allocated to the InnoDB buffer

✉ Ke Zhou
k.zhou@hust.edu.cn

Ji Zhang
jizhang@hust.edu.cn

Guoliang Li
liguoliang@tsinghua.edu.cn

Yu Liu
liu_yu@hust.edu.cn

Ming Xie
reganxie@tencent.com

Bin Cheng
bencheng@tencent.com

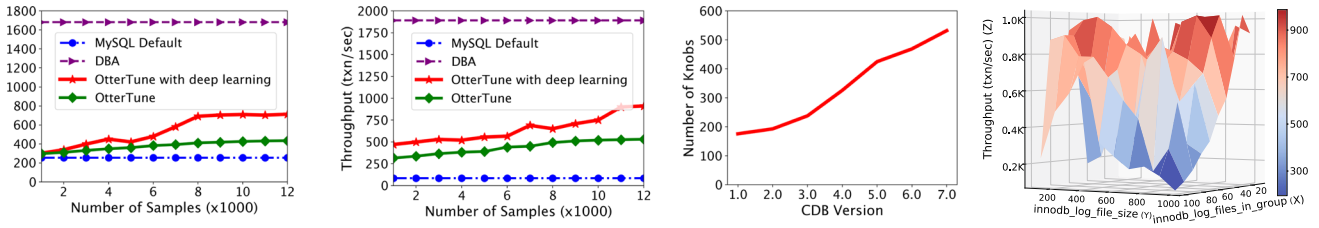
Jiashu Xing
flacroxing@tencent.com

[1] Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China

[2] University of Amsterdam, Amsterdam, The Netherlands

[3] Department of Computer Science, Tsinghua University, Beijing, China

[4] Tencent Inc., Shenzhen, China



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

Fig. 1 **a** and **b** show the performance of OtterTune [55] and OtterTune with deep learning over number of samples compared with default settings (MySQL v5.6) and configurations generated by experienced DBAs on CDB[2] (developed by the company Tencent). **c** shows the number of tunable knobs provided by a CDB in different versions. **d** shows the performance surface of a CDB (Read-Write workload of Sysbench, physical memory = 8 GB, disk = 100 GB)

pool. The tunable knob *table_open_cache* denotes the number of cached open tables for all threads which shows the number of tables that can stay open at the server in every parallel session. 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 the challenge that they have to tune cloud database systems for a large number of users with a limited number of expensive DBA experts. Thus, developing effective systems to accomplish automatic parameter 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 [67] and learning-based methods [15,40,55]. The search-based methods, e.g., BestConfig [67], 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 process whenever a new tuning request arrives, and thus fail to utilize knowledge gained from previous tuning efforts.

The learning-based methods, e.g., OtterTune [55], utilize machine learning (ML) techniques to collect, process and analyze knobs, and recommend possible settings by learning a 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 later 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, workloads, storage media type, CPU model and database type. It is hard to reproduce all conditions and accumulate high-quality samples. As shown in Figs. 1a and 1b, without high-quality samples, OtterTune [55] 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 Fig. 1c. One cannot optimize the knob settings in a high-dimensional continuous space by just using a 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 [55]. Moreover, the knobs are in continuous space and have unseen dependencies. As shown in Fig. 1d, 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, due to the flexibility of a cloud environment, users often change the hardware configuration, such as adjusting the memory size, disk capacity and storage media type. According to statistics from Tencent, 1800 users have made 6700 adjustments in half a year. In this case, conventional machine learning approaches have poor adaptability, and need to retrain the model to adapt to the new environment.

In this paper, we design an end-to-end automatic cloud database tuning system called `CDBTune`$^+$ using deep reinforcement learning (RL). `CDBTune`$^+$ uses the reward function in RL to provide feedback for evaluating the performance of a cloud database, and provides 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 trial-and-error method to enable utilizing a few samples to tune knobs for achieving higher performance, which alleviates the burden of collecting a large number of samples in the initial stage of modeling, and is more in line with a DBA's judgments and tuning

actions in real-world scenarios. CDBTune+ utilizes deep deterministic policy gradient method to find the optimal configurations in a continuous space, which solves the problem of quantization loss caused by regression in existing methods. Besides, we propose effective techniques to improve the training and tuning efficiency of CDBTune+ for practical use in a cloud environment. We conducted extensive experiments under 7 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 to existing tuning tools and DBA experts. Besides, CDBTune+ has a high adaptability so that the performance of CDB[1] deployed on configurations recommended by CDBTune+ will not decline even though the environment (e.g., memory, disk, workloads and storage media type) changes. Note that the CDB in our paper is a redeveloped cloud database whose kernel is MySQL, MongoDB or Postgres by Tencent which is a cloud database hosting service that combines high performance, high availability, high security, high scalability and ease of use.

In this paper, we make the following contributions:

(1) To the best of our knowledge, we design the first end-to-end automatic database tuning system that uses deep RL to learn and recommend configurations for databases.
(2) We adopt a trial-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) We propose a prioritized experience replay in our CDBTune+ to accelerate the convergence of our model and explore how to reduce the time-consuming restart time in order to provide users with a better experience in practical use.
(6) Our experimental results demonstrate that CDBTune+ could recommend knob settings that greatly improved performance, compared with the state-of-the-art tuning tools and DBA experts. Furthermore, we found that our system adapts well to a cloud environment (e.g., memory size, disk capacity, workloads and storage media type) changes. Our system is open-sourced and publicly available on Github[2].

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

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

2 System overview

In this section, we present our end-to-end automatic cloud database tuning system CDBTune+, which uses deep RL. We first introduce the general approach of CDBTune+ (Sect. 2.1) and then present the architecture of CDBTune (Sect. 2.2).

2.1 CDBTune+ working mechanism

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

2.1.1 Offline training

We first briefly introduce the basic idea of the model training (and more details will be discussed in Sects. 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 and r is the performance when processing q (including throughput and latency). We use the SQL command "show status" to obtain the state s, which is a common command that a DBA uses to understand the state of database. The state metrics contain the statistical information of the CDB, which describe the current state of the database; We refer to them as *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 Sect. 2.2.4). Note there is almost no cost to collect training data, because the collection interval is on the order of seconds.

Training model Because the DBMS configuration tuning problem that aims to find the optimal solution in continuous space is *NP*-hard [55], we use deep RL as the training model. RL adopts a trial-and-error strategy to train the model, which explores more optimal configurations that a DBA might never try, reducing the possibility of falling into a local optimum. Note that the RL model is trained offline once, and will subsequently be used to tune the database knobs for each tuning request from database users. The details of the model training will be introduced in Section 3.

Training data generation There are two ways to collect the training data. (1) ***Cold Start.*** Because of the lack of historical training data at the beginning of the offline training process, we utilize standard workload testing tools (such as

Sysbench[3], TPC-MySQL[4] and other query generator) to generate a set of query workloads. Then for each query workload q, we execute it on the CDB and generate the initial quadruple. Next, we apply the trial-and-error strategy described previously to train on the quadruple and generate 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 the configurations CDBTune$^+$ recommends. The gradual addition of real user behavior data to the training process allows CDBTune to 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$^+$ (analogous to 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 quickly in about 150 s, gets the current knob configuration a, and executes the query workload in the CDB to generate the current state s and performance r. Next it uses the model obtained by offline training to conduct online tuning. Eventually, the 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 the 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 independent and identically distributed (i.i.d.) hypothesis between samples in deep learning. Specifically, one of the most common assumptions in many deep learning approaches is that the given data samples are realizations of i.i.d. random variables. In deep reinforcement learning, in general, successive states (actions and rewards) are highly correlated. An "experience replay" buffer (see Sect. 5.1) was used in DDPG architecture to avoid training the neural network which represents the Q function, with correlated (or non-independent) sample [16]. 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 (which is also done during offline training). Note that CDBTune needs to fine-tune the pre-trained model in order to adapt to the real user workload. There are mainly two differences between online tuning and offline training. On the one hand, we no longer utilize the simulated data. Instead, we replay the user's current workload (see Sect. 2.2.1) to conduct stress testing on CDB in order to fine-tune (commonly used in ML, same with offline training just decrease the model learning rate) the model. On the other hand, the tuning terminates if the user obtains a satisfying performance with the improvements over the initial configuration or the number of tuning steps reaches the predefined maximum. In our experiments, we set the maximum number to 5.

Because restarting the DBMS is undesirable for users on the primary database instance in a cloud environment which will take more costs, we employ this tuning processing in the backup databases. In general, they have same the hardware as the primary databases and are therefore representative of them. We tailored the technical improvements (this will take extra costs to help our CDBTune$^+$ works which are beyond the scope of our main work here) based on the general cloud database backup technologies (e.g., via log shipping) to ensure that the secondary copy can accurately reflect the performance characteristics of the primary. Note that although the OtterTune does not state that it relies on a backup database or a primary one, we think it directly employs the primary database for tuning since they do not take into account the cost of restarting into consideration when choosing configurations but as future work [55]. Besides, CDBTune$^+$ also works well on the primary since our method is not limited to whether to use it on the backup or primary database, considering that working on the primary one causes unacceptable restarting to be impractical for users, we decide to deploy it on backup databases in our case. Of course, many low-cost cloud database services only have one database process and no secondaries or the general cloud database backup technology does not guarantee that the secondary copy accurately reflects the performance features of the primary one which makes it is an interesting and challenging question for future work.

2.2 System architecture

Figure 2 illustrates the architecture of CDBTune$^+$. The dotted box on the left represents the client, where users send their tuning requests to the server through the local interface. The other dotted box represents our tuning system in the cloud, in which the controller of the distributed cloud platform coordinates the client, the CDB and CDBTune$^+$. When the user initiates a tuning request or the DBA initiates a training request via the controller, the workload generator conducts stress testing on the CDB 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 in the CDB.

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

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

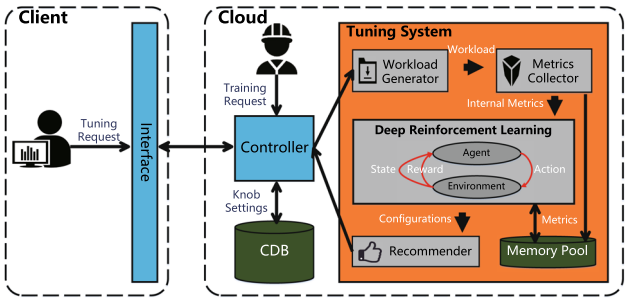


Fig. 2 System architecture

2.2.1 Workload generator

Our workload generator mainly takes care of two tasks: generating the standard testing workload and replaying the current user's real workload. Due to a lack of samples (historical experience data) during initial training, we can utilize standard workload testing tools such as Sysbench/TPC-MySQL combined with the trial-and-error manner of RL to collect simulated data, avoiding to strongly depend 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 logs from a given period 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 better configurations.

2.2.2 Metrics collector

When tuning a CDB upon a tuning request, we will collect and process the metrics data which captures the aspects of the CDB's runtime behavior in a certain time interval. Since the 63 metrics represent the current state of the database and are fed to the deep RL model in the form of vectors, we need to compute representative features from them. For example, we take the mean value of a state value in a certain time interval and compute the difference between the 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 the reward (which denotes how the performance of the current database will change after performing the corresponding knobs change in Sect. 4.2). Note that the DBA also collects the average values for these metrics by executing the "show status" command during the tuning tasks. Although these values may change over time, the average value can describe the database state well. This method is intuitive and simple, and we also experimentally validate its effectiveness.

We also investigate other methods. For example, we replace the average value by taking the maximum and minimum values of metrics in a period of time, which just grasp the local state of database. Experimental results show that the maximum and minimum values do not work as well as the mean value due to their lack of accurately grasping the database state (see Sect. 6.1.6). Last but not least, we would like to highlight that our usage of the "show status" command to get the database states does 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 parameter setting commands, and sends a configuration modification request to the controller. After acquiring the DBA's or user's approval, the controller deploys these configurations to the CDB 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 types of information: the state of the current database s_t (in the form of vectorized internal metrics), the reward value r_t calculated by the reward function (which will be introduced in Sect. 4.2) via external metrics, the knobs to be set on the database a_t, and the database's state vector after applying the configurations s_{t+1}. A sample can be represented 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; we therefore refer to it as experience replay memory.

3 RL in CDBTune+

We introduce RL to simulate the trial-and-error method that the DBA adopts, and to overcome the shortcomings caused by regression. RL originates from trial-and-error learning in animal learning psychology and is a key technology to approach the *NP*-hard problem of database tuning in continuous parameter space. We clarify our notation in Table 1.

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 want our model to learn well with limited samples during initial training and to simulate the DBA's train of thought as much as possible. Therefore, we tried the

Table 1 Notations

Variables	Descriptions	Mapping to CDBTune$^+$
s	State	Internal metrics of the DBMS
a	Action	Tunable knobs of the DBMS
r	Reward	The performance of the DBMS
α	Learning rate	Set to 0.001
γ	Discount factor	Set to 0.99
ω	The weights of the 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	–

RL method. At the beginning, we tried classic Q-learning and DQN models in RL, but both of these methods failed to solve the problems incurred by our high-dimensional space problem (database state, knobs combination of knobs) and continuous actions (continuous knobs). Eventually, we adopt the policy-based Deep Deterministic Policy Gradient approach which overcomes these shortcomings effectively. In addition, the design of the reward function (RF) is vital (as the "soul" of RL), 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 effective and efficient. RL uses the exploration & exploitation strategy [27] to potentially explore more optimal configurations that the a DBA would never have tried. For example, when training our model, we randomly output a set of recommended configurations with a small probability (e.g., 0.1) and follow an output according to the network learning strategy with a large probability (0.9). This allows us to explore the unknown tuning space with a small probability so as to reduce the possibility of falling into a local optimum. Although the exploration in RL will not result in arbitrarily bad performance from the users' side (because we employ the training and tuning process first to backup databases), we are working hard to explore techniques that could constrain the recommended configuration to a "safe" range. This is an interesting and challenging question for future work.

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 Fig. 3, we show an interaction diagram of the six key elements in RL and the correspondence between the six elements and database configuration tuning.

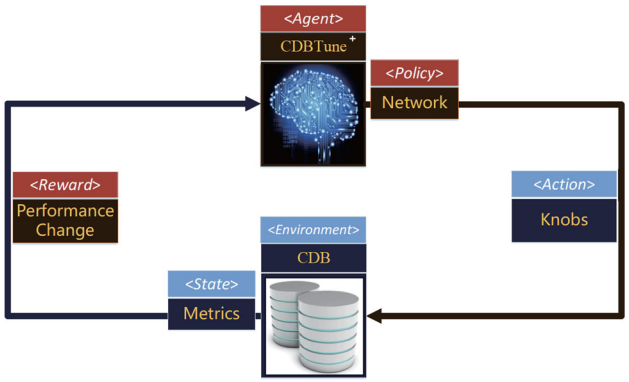


Fig. 3 The correspondence between RL elements and CDB configuration tuning

Agent The agent can be seen as the tuning system CDBTune$^+$ which receives a reward (i.e., the performance change) and a state from the CDB, and updates the policy guiding the system to adjust the knobs for getting a higher reward (higher performance).

Environment The environment is the tuning target, specifically an instance of a CDB.

State The state denotes the current state of the agent, i.e., the 63 metrics. Specifically, when CDBTune$^+$ recommends a set of knob settings and the CDB applies 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 the CDB. In general, we describe the state at time t as s_t.

Reward The reward is a scalar r_t which denotes the difference between the performance at times t and that at $t-1$ or the initial settings, i.e., the performance change after/before the CDB applied the new knob configurations that CDBTune$^+$ recommended at time t.

Action An action originates from the space of knob configurations, which is often described as a_t. An action here corresponds to a knob tuning operation. The CDB applies the corresponding action according to the latest policy under the corresponding state of the CDB. Note that an action might increase or decrease several tunable knobs' values at a time.
Policy The policy $\mu(s_t)$ defines the behavior of CDBTune$^+$ at a certain specific time and in a certain environment, It maps a state to an action. In other words, given a CDB state, if an action (i.e., a knob tuning) is called, the policy outputs the next state by applying the action to the original state. The policy here is a deep neural network, which contains 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 the corresponding deep neural network in Sect. 4.
RL-Based learning. The learning process of DBMS configuration tuning in RL is as as follows. The 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 a deep neural network (policy) whose input is the database state and whose output are the recommended configurations corresponding to the state. When applying the recommended configurations to the CDB, the current state of the database will change, which is reflected in the metrics. The internal metrics can be used to measure the runtime behavior of a database corresponding to the state in RL, while external metrics can evaluate the performance of a database for calculating the corresponding feedback reward value in RL. The agent will update its network (policy) according to these two pieces of feedback in order to recommend better performing knobs. This process iterates until the model converges. Ultimately, the most appropriate knob settings will be exposed. Note that we think CDBTune$^+$ is not database specific if the objective optimization system (e.g. a storage system, Spark, Hive and Tomcat) can be abstracted to the six elements mentioned above. This is an interesting question for future work to verify.

3.3 RL for tuning

RL makes a policy decision through the interaction process between agent and environment. In contrast to supervised learning or unsupervised learning, RL depends on accumulated rewards, rather than labels, to perform training and learning. The goal of RL is to optimize its own policy based on the reward of the environment by interacting with the environment and achieving higher rewards by acting according to the updated policy. The agent is able to discover the best action through a trial-and-error strategy by either exploiting current knowledge or exploring unknown states. The learning of our model follows the two rules that the action depends on the policy, and that the policy is driven by the expected rewards of each state. RL can be divided into two categories: value-based method and policy-based methods. 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 a concrete 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.

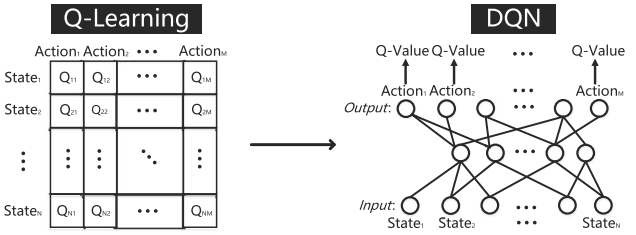


Fig. 4 Difference between Q-Learning and DQN

Q-Learning It is worth noting that Q-Learning [34] is one of the most popular value-based RL methods, at whose core is the calculation of Q-tables, which are defined as $Q(s, a)$. The rows of the Q-tables contain the Q-value of the states while the columns of the Q-table represent actions, which measures how beneficial it will be if the current state is followed by this action. $Q(s, a)$ is iteratively defined as follows:

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

The basis for updating the 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 not well suited to solve problems with 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 a database in a cloud environment are also complex and diverse. For example, suppose that each inner metric value ranges from 0 to 100 and its value is discretized into 100 equal bins. Then 63 metrics will then have 100^{63} states. As a result, applying Q-Learning to database configuration tuning is impractical.
DQN Fortunately, the Deep Q Network (DQN) [36] method is able to solve the problems mentioned above effectively. DQN uses neural networks rather than Q-tables to evaluate the Q-value, which fundamentally differs from Q-Learning (see Fig. 4). In DQN, the input are states while the output are the Q-values of all actions. Nevertheless, DQN still adopts Q-Learning to update the 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 the neural network in DQN.

Unfortunately, DQN is a discrete-oriented control algorithm, which means that the actions it outputs are discrete. Taking the maze game for example [66], only four directions of output can be controlled. However, knob combinations in a database are high-dimensional and the values for many of them are continuous. For instance, if we use 266 (the maximum number of knobs that the DBA uses to tune a CDB) continuous knobs which range from 0 to 100. If each knob range would be discretized into 100 intervals, there would be 100 values for each knob. Thus there would be 100^{266} actions (knob combinations) in total for DQN. Further, if we increase the number of knobs or decrease the learning interval, the scale of outputs would increase exponentially. Thus, neither Q-Learning nor DQN can solve the issue of database tuning. Thus we introduce the policy-based RL method DDPG to address this issue in Sect. 4.

4 DDPG for CDBTune$^+$

The "Deep Deterministic Policy Gradient" (DDPG) [33] algorithm is a policy-based method for RL. DDPG is a combination of the DQN and an 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 having to compute and store the corresponding Q-values for all actions, as DQN has to. Therefore, DDPG can learn the policy with high-dimensional states and actions, in our case with the internal metrics and knob configurations. As a consequence, we choose DDPG in CDBTune$^+$ for our use case.

In this section, we first introduce the policy-based RL method DDPG and the prioritized experience replay in DDPG, then describe our custom reward function, and finally summarize the advantages of applying RL to the database tuning problem.

4.1 Deep deterministic policy gradient with prioritized experience replay

We illustrate DDPG for CDBTune in Fig. 5. When utilizing DDPG in CDBTune$^+$, firstly, we regard the CDB instance to be tuned as the 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 applying a_t to the instance. Analogous to most policy gradient methods, DDPG has a

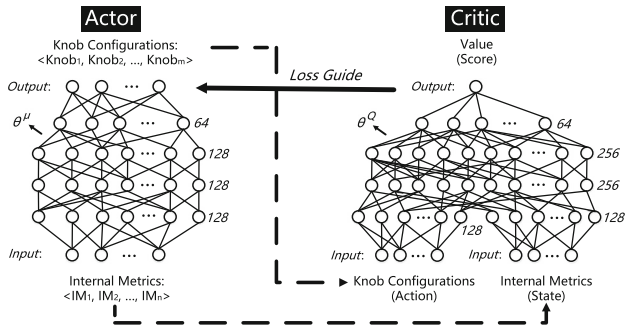


Fig. 5 DDPG for CDBTune$^+$. It consists two parts: an actor network and a critic network

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 an actor). The critic function $Q(s_t, a_t|\theta^Q)$ of the network (where θ^Q denotes the learnable parameters) aims to represent the value (score) for a specific action a_t and state s_t, which guides the learning of actor. Specifically, the critic function helps to evaluate the knob settings generated by the actor according to the current state of the instance. Inheriting the insights from the 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 replay memory, we apply the Q-learning algorithm and minimize the training objective:

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

where

$$y = r(s_t, a_t) + \gamma Q^\mu(s_{t+1}, \mu(s_{t+1})|\theta^Q) \quad (4)$$

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

$$\begin{aligned} \nabla_{\theta^\mu} J &\approx \mathbb{E}[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)}] \\ &= \mathbb{E}[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_t}] \end{aligned} \quad (5)$$

The algorithm contains seven main steps, which are summarized in pseudo code in Algorithm 1. Note that we use the prioritized experience replay (PER) [43] method to accelerate our model convergence in online training (detailed see Sect. 5.1).

Algorithm 1 Deep deterministic policy gradient (DDPG) with Prioritized Experience Replay (PER)

1: Sample a transition (s_t, r_t, a_t, s_{t+1}) based on prioritized experience replay in DDPG 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:

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

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

Step 2 We feed s_{t+1} into the actor network and output the knob settings a'_{t+1} to be executed at the 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 the Q-Learning algorithm, V_{t+1} is multiplied by the discount factor γ and added to the value of the reward at time t. 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 squared difference between V'_t and V_t and update the parameter θ^Q of the critic network by gradient descent.

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

To make it easier to understand and implement our algorithm, we elaborate the network structure and specific parameter values of DDPG in Table 2. Note that we also discuss the impact of the recommended configurations by different networks on the system's performance in Sect. 6.1.5.

Remark. Traditional machine learning methods rely on a massive amount of training samples to train the model. In contrast to that, we adopt the trial-and-error method to make our model generate diversified samples and learn via deep reinforcement learning, which only requires a limited amount of samples. We summarize the advantages of this approach. Firstly, for solving traditional game problems, we cannot predict how the environment will change after taking an action, because the environment of the game is random (for example, in Go, we do not know what the opponent will do next). However, in our DBMS tuning scenario, after a configuration is executed, the database (environment) will not randomly change after a configuration changes, due to the dependencies between knobs. Because of this, with relatively few samples, it is easier to learn DBMS tuning models than game problems with an adversary. Secondly, our CDBTune$^+$ model, only requires few input and output dimensions (63 and 266), which allows the network to efficiently converge without too many samples. Thirdly, RL also requires diverse samples, not only massive samples. For example, RL solves the game problems by processing each frame of the game screen to form the initial training samples. The time interval for each frame is very short, leading to a high redundancy of the training images. On the contrary, for DBMS tuning, we will change the parameters of database and collect the performance data. These data are diverse in our learning process, which allows us to constantly update and optimize the performance. Lastly,

Table 2 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

coupled with our efficient reward function, our method performs effectively with a small number of diverse samples (which will be evaluated in Sect. 6.3.1).

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 directly predict the values for all tunable knobs at the same time without considering the Q-value of a specific action and state.

4.2 Reward function

The reward function is vital in RL, as it determines the feedback information between the agent and environment. We aim for a function that simulates a DBA's empirical judgment of a real environment in the tuning process.

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

(1) Suppose that the initial performance of the DBMS is D_0 and the final performance achieved by the DBA is D_n.
(2) The DBA tunes the knobs and the performance changes to D_1 after the first tuning step. Then the DBA computes the performance change $\Delta(D_1, D_0)$.
(3) At the ith tuning iteration, the 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 the DBA aims to improve the performance through the tuning. However, the DBA cannot guarantee that D_i is better than D_{i-1} at every iteration. To this end, the 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 aim to mimic the tuning method of DBAs, which not only considers the change of performance compared to the previous step but also to the initial state (the time when the decision to tune the database was made). Formally, let r, T and L denote reward, throughput and latency. Especially, T_0 and L_0 respectively denote the throughput and latency before tuning. We design the reward function as follows.

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

$$\Delta T = \begin{cases} \Delta T_{t\to 0} = \dfrac{T_t - T_0}{T_0} \\ \Delta T_{t\to t-1} = \dfrac{T_t - T_{t-1}}{T_{t-1}} \end{cases} \tag{6}$$

$$\Delta L = \begin{cases} \Delta L_{t\to 0} = \dfrac{-L_t + L_0}{L_0} \\ \Delta L_{t\to t-1} = \dfrac{-L_t + L_{t-1}}{L_{t-1}} \end{cases} \tag{7}$$

According to Eqs. (6) and (7), we design the reward function below:

$$r = \begin{cases} ((1+\Delta_{t\to 0})^2 - 1)|1+\Delta_{t\to t-1}|, \Delta_{t\to 0} > 0 \\ -((1-\Delta_{t\to 0})^2 - 1)|1-\Delta_{t\to t-1}|, \Delta_{t\to 0} \le 0 \end{cases} \tag{8}$$

Δ can refer to the performance change of latency L or throughput T. As 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 when designing the reward function. We therefore set r to 0 if the result of Eq. (8) is positive and $\Delta_{t\to t-1}$ is negative.

We calculate the reward for throughput r_T and latency r_L according to Eq. (8). We combine these two rewards with the corresponding weights C_L and C_T, where $C_L + C_T = 1$. Note that these weights can be set based on user preferences. We have r to denote the sum of rewards of throughput and latency:

$$r = C_T * r_T + C_L * r_L \tag{9}$$

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

Note that other reward functions can be integrated into our system as well. We evaluate our designed reward function for the training and tuning process, by comparing it with three other typical reward functions in Sect. 6.3.1. Moreover, we explore how the two weights C_L and C_T will affect the performance of the DBMS in Sect. 6.3.2.

4.3 Advantages

We briefly summarize the advantages of our method. (1) **Limited Samples.** In the absence of high-quality empirical data, accumulating experience via a trial-and-error method reduces the cost of data acquisition. We have our model generate diverse samples and learn towards an optimizing direction by using deep RL which only requires a limited

amount of samples to achieve great effects (see Sect. 6). (2) **High-dimensional Continuous Knobs Recommendation.** The DDPG algorithm recommends better configurations in high-dimensional continuous space than simple regression methods, which are for example used by OtterTune (see Sect. 6.2.2). (3) **End-to-End Approach.** Our end-to-end approach reduces the potential for errors caused by multiple segmented tasks and improves the precision of recommended configuration. Moreover, the importance of different knobs is treated as an abstract feature which is implicitly learned by our deep neural network instead of us having to apply an extra method to rank the importance of different knobs (see Sect. 6.2.2). (4) **Reducing the Possibility of a Local Optimum.** CDBTune$^+$ may not find the global optimum, but RL adopts the well-known exploration & exploitation strategy, which is designed to efficiently explores configurations that a DBA may never try, thereby reducing the possibility of getting stuck in a local optimum (see Sect. 6.2.3). (5) **High Adaptability.** In contrast to supervised or unsupervised learning, RL has the ability to learn as much as possible in a reasonable direction from experience rather than from given examples, with a much lower dependency on labels or training data and shows a much higher adaptability to different workloads and hardware configurations in a cloud environment (see Sect. 6.4).

5 Improving the efficiency of CDBTune$^+$

In this section, we explore for improving the efficiency of CDBTune$^+$ in practical usage. We first describe the prioritized experience replay (PER) method in DDPG, and then introduce how to reduce the restart time of a CDB.

5.1 Prioritized experience replay in DDPG

Successful attempts obtained during reinforcement learning are rare, resulting in an imbalance of positive and negative transitions. If a transition is randomly sampled in step 1 of Algorithm 1, failed attempts will be sampled with a high probability, which results in low efficiency of learning. How can we effectively prioritize the transitions we need to learn in DDPG? Prioritized experience replay (PER) is a method that is widely used in DQN. It is based on the idea to more frequently replay experiences associated with very successful attempts or extremely awful performance. Therefore, we adopt this method in DDPG. The so-called temporal difference error (TD-error) is used to update the estimate of the critic function $Q(s, a)$ in DDPG. The computation of the TD-error TD_m of a transition m is given as:

$$TD_m = V_m - V'_m \tag{10}$$

V_m is the Q-value/score of the current state and V'_m is the estimated oneThe TD-error value is the loss of the critic network (which can be described as "how far the Q-value is from its next bootstrap estimation") which reflects a correction for the estimation and may implicitly reflect to what extent an agent can learn from the experience. The larger the TD-error value is, the more space for improvement is still present in the prediction accuracy of the model, and the more important it is to learn from this transition. Replaying these transitions more frequently will help CDBTune$^+$ to gradually realize the true consequence of the wrong behavior in the corresponding states as well as avoid applying the wrong behavior in these conditions again, which can improve the overall database tuning performance. However, using the TD-error as the metric to sample the transitions will result in some transitions with very small TD-error values that are never sampled for learning. Therefore, we define the probability of the sampled transition m via the SoftMax:

$$P(m) = \frac{p_m^\epsilon}{\sum_k p_k^\epsilon} \tag{11}$$

where $p_m = (\text{rank}(m))^{-1}$, and $\text{rank}(m)$ denote the rank of the transition m in the experience replay memory with absolute TD-error being the criterion. Besides, the parameter ϵ is the parameter which controls to what extent the prioritization is used. (Note that setting ϵ to 1 corresponds to random sampling). This sampling strategy ensures that even transitions with a low TD-error have a chance of being sampled, which increases the diversity of sampled transitions and regularizes the model. Last but not least, since we tend to more frequently replay the transitions with a high TD-error, we violate the assumption that the states in reinforcement learning are accessed randomly. This may lead the convergence problems for CDBTune$^+$. In order to solve this problem, importance-sampling weights are used in the computation of weight changes:

$$W_m = \frac{1}{N^\beta * P(m)^\beta} \tag{12}$$

where N is the number of transitions and β is the parameter, which controls to what extent the correction is used. According to the above description. The Prioritized Experience Replay method in DDPG (PER-DDPG, Step 1 in Algorithm 1) used in our CDBTune$^+$ training process (and following our outlined approach) is shown in Algorithm 2. We also have evaluated the effect of this method on training CDBTune$^+$ in Sect. 6.1.1.

Algorithm 2 Prioritized Experience Replay in DDPG (PER-DDPG)

1: Store transition (s_t, r_t, a_t, s_{t+1}) in experience replay memory and initialize the minibatch B.
2: **for m = 1 to B do**
3: Sample transition m with probability $P(m)$.
4: Compute the corresponding importance-sampling weight W_m.
5: Compute the TD-error TD_m.
6: Update the transition m priority based on the absolute TD-error $|TD_m|$.
7: **end for**

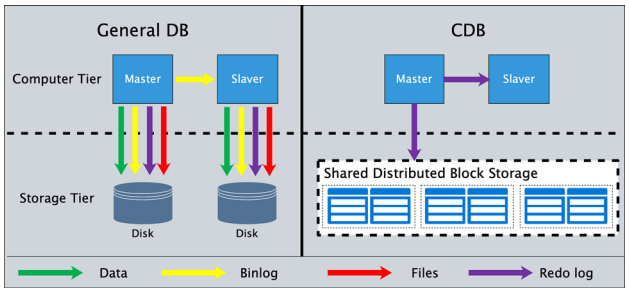


Fig. 6 Architectural differences between CDBs and general databases. CDBs use shared distributed block storage instead of local storage

5.2 Instant restart of a CDB

Changes in some of the tunable knobs in a CDB require a restart the database system. The restart time ranges from a few minutes to ten minutes because most database systems follow the classic Algorithms for Recovery and Isolation Exploiting Semantics (ARIES) [37], which typically create checkpoints periodically and store this information in a log file. When a database needs to restart, the system first needs to conduct the redo recovery from the last checkpoint to restore the data to the state before the restart (storage tier), then apply the undo recovery based on the undo log records to complete uncommitted transactions (compute tier). Note that a large amount of data I/O, file I/O, binlog I/O and redo log I/O operations are required to interact between the two tiers in this process. Therefore, the restart of a database system is a very time consuming operation.

To solve this problem, cloud databases draws on the idea of "The log is the database" from Aurora [56], PolarDB [8] and AnalyticDB [65] The differences between cloud databases and general database are shown in Fig. 6. CDBs differ in three characteristics from a general database: (1) they decouple the storage tier from the compute tier; (2) they offload all other types of I/O (data, file and binlog) except for log I/O; (3) CDBs use shared distributed block storage instead of local storage. Therefore, in contrast to the restarting process of general databases like MySQL where the redo recovery in the storage tier follows the undo operation in compute tier, the decoupled design between compute tier and storage tier in CDBs allows the storage tier to continuously construct the latest version of the data in parallel with the compute tier in an asynchronous manner. Besides, CDBs offload most of the I/O access in this architecture, minimizing the bandwidth between the compute and storage tiers. Thus, CDBs achieve a nearly instant restart, where the restart time is mostly around 5 s. The reduction of the restart time significantly reduces the waiting time (online tuning) for cloud database users of CDBTune$^+$ and thus provides them with a better experience in practical use (detailed see in Sect. 6.1.3). In addition, if the user is not sensitive to the online tuning time, CDBTune$^+$ can try more steps in the same online tuning time and might find better configurations to achieve higher performance (detailed see Sect. 6.1.4).

6 Experimental study

In this section, we evaluate the performance of CDBTune$^+$ and compare it to existing approaches. We first show the execution time of our model in contrast to existing baselines, then evaluate the impact of prioritized experience replay (PER) in CDBTune$^+$ and show the performance of varying tuning steps, neural networks and types of metric data. Secondly, we compare the performance of CDBTune$^+$ with BestConfig [67], OtterTune [55], and 3 DBA experts who have been engaged in tuning and optimizing DBMSs for 12 years at Tencent. Additionally, we evaluate CDBTune$^+$ on other databases and storage media. Finally, we evaluate our custom reward function, and verify the adaptability of the model under different conditions.

Workload. Our experiments are conducted using three kinds of benchmark tools: Sysbench, MySQL-TPCH and TPC-MySQL. We carry out the experiments with 7 workloads consisting of read-only, write-only and read-write workloads of Sysbench, TPC-H[5] workloads, TPC-C workloads, YCSB workloads and JOB workloads, which are similar to existing work. Under Sysbench workloads, we set up 16 tables each of which contain about 200 K records (about 8.5 GB) and we set the number of threads to 1500. For TPC-C (OLTP), we select a database consisting of 200 warehouses (about 12.8 GB) and set the number of concurrent connections to 32. The TPC-H (OLAP) workloads contain 16 tables (about 16 GB). For YCSB (OLTP), we generate 35GB data using 50 threads and 20M operations. For JOB workloads, we leverage 21 tables (about 13 GB). We abbreviate read-only, write-only and read-write workloads of Sysbench as RO, WO and RW respectively. We denote online tuning using a model trained on another condition via the expression M_{training

[5] http://www.tpc.org/tpch

Table 3 The training data for CDBTune$^+$ and OtterTune

#Knobs	Samples of CDBTune$^+$	Samples of OtterTune
16	208	208 (CDBTune$^+$)+10 (DBA)
65	469	469 (CDBTune$^+$)+24 (DBA)
266	1530	1530 (CDBTune$^+$)+75 (DBA)

Table 4 Database instances and hardware configuration

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

condition}→{tuning condition}. For example, when we use 8 GB RAM as a training setting and apply the resulting model for online tuning on 12 GB RAM, and then we denote this as $M\_8G \rightarrow 12G$.

Training data It is hard to collect a large amount of training data to adequatly represent the tuning experience of a DBA. We show the data for training CDBTune$^+$ and comparing to OtterTune [55] in Table 3. We utilize all the accumulated DBA experience data as well as the training data for CDBTune$^+$ to train OtterTune. The proportion of these two datasets is about 1:20. For example, when recommending the configuration for 266 knobs, CDBTune$^+$ collects 1500 samples; OtterTune will additionally use 75 historical samples and the samples of tuning data of a DBA. Note that the DBA's samples and the samples CDBTune$^+$ adopted are different, where CDBTune$^+$ collected the sample as mentioned in Sect. 2.1.1. The samples from DBAs contain two parts: workload (internal metrics) and current configuration that DBA tuned (Note that these are the same as the samples OtterTune and Bestconfig used).

Setting Our CDBTune$^+$ is implemented using PyTorch[6] and Python libraries including the scikit-learn library[7]. All the experiments are run on Tencent's cloud servers with a 12-core 4.0 GHz CPU, 64 GB RAM and a 200 GB Disk. We use 10 types of CDB instances in the evaluation and show their hardware configurations in Table 4. The difference between them is mainly reflected in the memory size, disk capacity and storage media type. 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 allow it to run for 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 DDPG with priority experience replay to accelerate the convergence (as described in Section 6.1.1), which increases the convergence speed by a factor of two (halving the number of iterations). We also adopt parallel computing (30 servers) which greatly reduces the offline training time (Note that we do not use parallel computing for online tuning).

6.1 Efficiency comparison

We first evaluate impact of the PER Method in CDBTune$^+$ and the execution time details of our method, then we compare with baselines and show the performance of varying the tuning steps, neural networks and the types of metrics data.

6.1.1 The impact of PER method in CDBTune$^+$

As mentioned in Sect. 5.1, the core idea of the prioritized experience relay (PER) is to more frequently replay impactful transitions in order to increase the efficiency and performance of modeling. For verifying the effectiveness of the PRE method during model training in CDBTune$^+$, we carry out the experiments using the Sysbench RW, RO and WO workload based on the database instance CDB-A. We record the tuning performance (throughput and latency) and the

[6] https://pytorch.org

[7] http://scikit-learn.org/stable

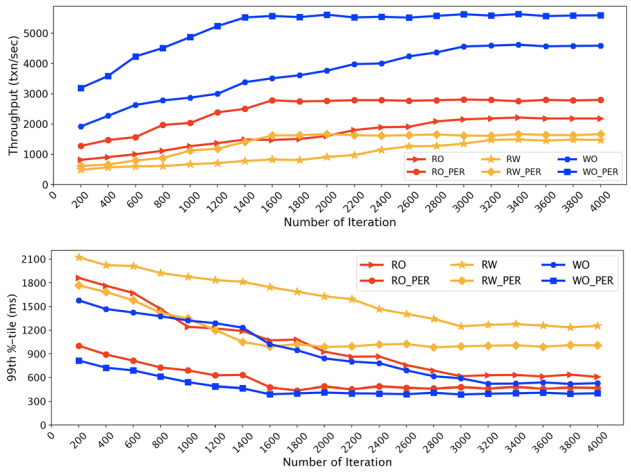


Fig. 7 Performance comparison for the Sysbench RO, RW and WO workload between CDBTune$^+$ with PER and without PER method in DDPG

final number of convergence for CDBTune$^+$ with or without PRE method in DDPG. The experimental results shown in Fig. 7. Combining DDPG with PER increases the convergence speed by a factor of two (about 1500 iterations with PER and 3200 iterations without PER) and results in a 10% to 20% improvement in tuning performance compared to training without PER. This demonstrates that DDPG with PER can prioritize more helpful transitions for the training of CDBTune$^+$ and increase the tuning performance and learning efficiency.

6.1.2 Breakdown of the execution time in CDBTune$^+$

In order to know how long a step takes in the training and tuning process, we record the average runtime of each step. This average runtime for each step is 3 min, which is mainly divided into 5 parts (excluding about 5 s to restart the CDB) as follows:

(1) *Stress Testing Time (152.88 sec)* Runtime of the workload generator for collecting the current metrics of the database.
(2) *Metrics Collection Time (0.86 ms)* Runtime of obtaining state vectors from internal metrics and calculating the 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 reading the database state to outputting recommended knobs.
(5) *Deployment Time (16.68 sec)* Runtime from outputting recommended knobs to deploying the configurations according to the CDB's API interface.

Offline training CDBTune$^+$ takes about 4.7 h for 266 knobs and 2.3 h 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$^+$ for 5 steps, so the online tuning time is 15 min.

6.1.3 Tuning efficiency comparison with baselines

The reduction of the restart time as mentioned in Sect. 5.2 significantly reduces the waiting time (online tuning) for cloud database users of CDBTune$^+$ and thus provides them with a better experience in practical use. Specifically, the minimum waiting period (5 trial steps, each step costs 3 min) was reduced to 15 min compared to the 25 min that the general database takes (a total reduction of 40%). Additionally, if the user is not sensitive to the online tuning time, CDBTune$^+$ can try more steps in the same online tuning time. CDBTune$^+$ will find better configurations to achieve higher performance for the users, as we will describe in Sect. 6.1.4. Note that the zero restart database technology can be explored in the future to improve the users' experience and the performance of the database tuning further. We just need to restart the database from the user side once because we employ the training and tuning process in the backup database. We compare the online tuning efficiency of CDBTune$^+$ with OtterTune [55], BestConfig [67] and a DBA. Note that only CDBTune$^+$ requires offline training. But it trains the model once and uses the same 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 5, for each tuning request, OtterTune takes 55 min, BestConfig takes about 250 min, the DBAs take 8.6 h, while CDBTune$^+$ takes 15 min. Note that we invite 3 DBAs to tune the parameters and select the best performance of their results which takes 8.6 h for each tuning request. (We have recorded 57 tuning requests from 3 DBAs, which in total took 491 hours.) It takes a DBA about 2 h to constantly execute the 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 a lot of experience and takes a lot of time. OtterTune adopts a simple GP regression, and the knobs recommended are not accurate. Therefore, it has to conduct 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 huge amount of trial time.

Table 5 Detailed online tuning steps and the time of CDBTune and other tools

Tuning tools	Total steps	Time of one step (mins)	Total time (mins)
CDBTune	5	3	15
OtterTune	5	11	55
BestConfig	50	5	250
DBA	1	516	516

6.1.4 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 pretrained model with a limited number of steps which we refer to as accumulated trial 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, knob settings that have never been tried in the past will be obtained with a very low probability only. These outliers may either degrade or improve performance to an unprecedented level. We decide to use 5 steps as an increment unit to observe the system's performance and record the recommended result corresponding to the optimal performance. We carry out experiments with CDB-A on three different Sysbench workloads respectively as shown in Fig. 8. Here, the horizontal coordinate represents the number of tuning steps before recommending configurations while the vertical ordinate represents the value of throughput or latency. We find that the standard model gradually adapts to the current workload through fine-tuning operations as the number of steps increases, which continuously improves 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 provides high efficiency. The main reasons for ensuring convergence in the first 5 steps include two aspects. Firstly, deep reinforcement learning can learn and obtain the decision-making experience in complex environments. Although hardware configuration and workloads may change, deep reinforcement learning can automatically extract the features that really work in different environments, thus simplifying the tuning difficulty, effectively making decisions, and adapting to the current environment. Secondly, for ensuring the accuracy of recommendations, we fine-tune the model with a limited number of steps and to allow it to quickly adapt to the current environment. Certainly, the user will get better configurations to achieve higher performance if accepting a longer tuning time. However, OtterTune exhibits diminishing returns with increasing number of iterations, which is caused by the characteristics of supervised learning and regression.

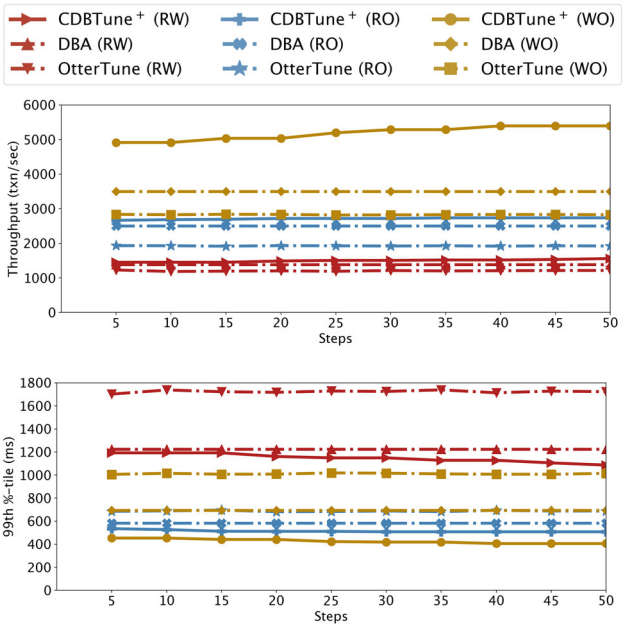


Fig. 8 Performance by increasing number of steps

6.1.5 Varying neural networks

In this section, we discuss the impact of recommended configurations by different network architectures on the system's performance when tuning 266 knobs. We mainly change the number of hidden layers and neurons in each layer of both the Actor and Critic network. The detailed parameters are displayed in Table 6. The performance decreases when the number of layers extends 5. This may result from the high complexity of our model due to the increasing number of layers, which leads to over-fitting. Moreover, when the number of hidden layers (of both two networks) is fixed, increasing the number of neurons in each layer seems to have little effect on the performance, but the number of required iterations increases a lot due to the higher complexity of the network. Based on the observations, 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 Fig. 5 in Sect. 4.1.

Table 6 Tuning performance varying neural network structure

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

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

Table 7 Comparison of the methods of processing metrics using instance CDB-A. The data in the table is (Throughput (txn/sec), Latency (ms))

Workload	Average	Peak	Throughput
RW	**(1559.31, 1086.57)**	(1276.55, 1648.75)	(1388.23, 1574.91)
RO	**(2736.52, 508.06)**	(2365.79, 598.43)	(2238.32, 581.39)
WO	**(5396.37, 405.74)**	(3976.62, 760.79)	(3688.12, 823.73)
TPC-C	**(1598.32, 2616.77)**	(1124.22, 3211.86)	(1315.76, 2916.77)
TPC-H	**(1633.17, 2545.22)**	(1237.83, 2876.01)	(1026.03, 3098.26)

6.1.6 Varying types of metric data

As mentioned in Sect. 2.2.2, we record the average value for internal metrics in order to describe the database state, and feed these into training our model. We also tried other methods. For example, we replace the average value by taking the maximum and minimum values of metrics in a period of time. The corresponding experimental result using database instance CDB-A is shown in Table 7. We can see that leveraging the average value results in better performance than using the other two values under 6 different workloads. This approach is inspired by the observation that DBAs often inspect the average value.

6.2 Effectiveness comparison

In this section, we evaluate the effect of varying numbers of knobs and discuss the performance of CDBTune, the DBA, OtterTune and Bestconfig with different workloads. For the database instance CDB-B, we record the throughput, latency, and number of iterations for the 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 operate on 266 tunable knobs (the maximum number of knobs that the DBA uses to tune a CDB). Note that we also provide more experiments on other databases and storage media.

6.2.1 Knobs selected by DBAs and OtterTune

Both the DBAs and OtterTune rank the knobs based on their importance to the performance of the database. We use their rankings to sort all our 266 knobs, and correspondingly select different numbers of knobs following the order to tune and compare different methods. Figures 9 and 10 show the performance of CDB-B under the TPC-C workload based on the rankings of the DBA and OtterTune respectively. We can see from the results that CDBTune$^+$ achieves better performance in all cases. Note that the performance of the DBA and OtterTune begins to decrease after the number of their recommended knobs exceeds a certain threshold. The main reason is that the unseen dependencies between knobs become more complex in a larger spaces, but the DBA and OtterTune cannot recommend reasonable configurations in such a high-dimensional continuous space.

6.2.2 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 step) and record the performance of CDB-B under the TPC-C workload after executing these configurations. As shown in Fig. 11, when 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 the selected knobs have a small impact on performance. Besides, after the number of knobs reaches a certain number,

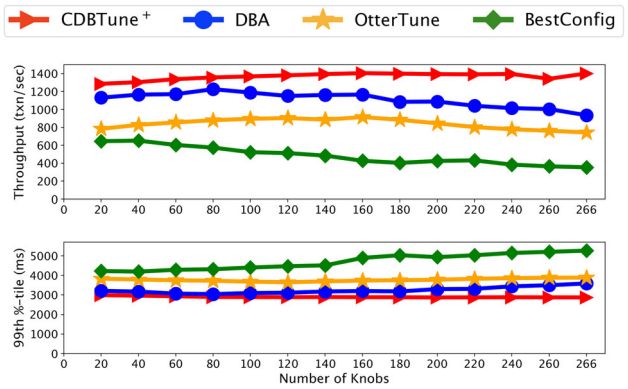


Fig. 9 Performance by increasing number of knobs (knobs sorted by DBA)

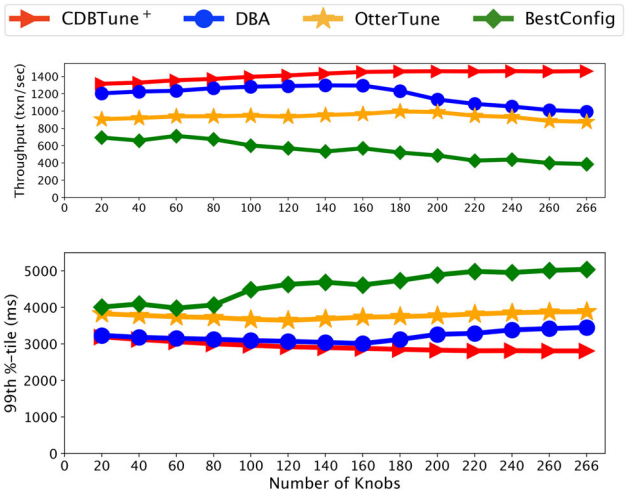


Fig. 10 Performance by increasing number of knobs (knobs sorted by OtterTune)

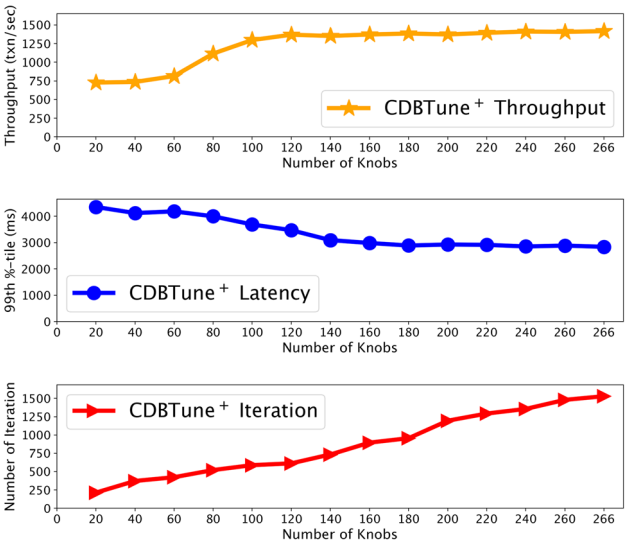


Fig. 11 Performance by increasing the number of knobs (knobs randomly selected by CDBTune+)

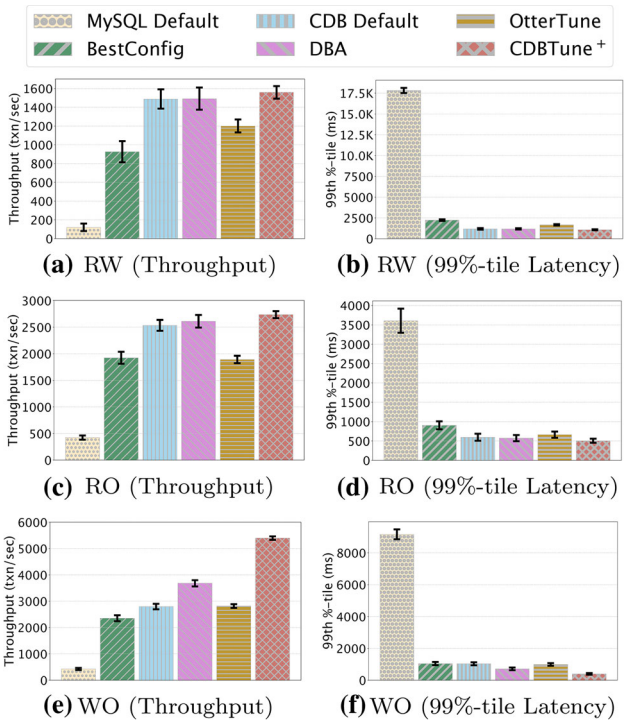


Fig. 12 Performance comparison for the Sysbench RW, RO and WO workloads between CDBTune+, MySQL default, BestConfig, CDB default, DBA and OtterTune

the performance tends to stabilize. In Sect. 6.2.1, CDBTune+ uses the knob importance order DBAs or OtterTune produced is just to verify that CDBTune+ can work well and achieves better performance than DBAs and OtterTune in all cases. Whereas, this experiment demonstrates that DBA and OtterTune separately rank the importance of knobs, but our CDBTune+ automatically completes this process as part of the learning of its deep neural network without an additional ranking step (as required by a DBA and OtterTune), which is also in line with our original intention of designing an end-to-end model.

In addition, the input and output of the network become larger as the number of knobs increases, having the effect that CDBTune+ takes more steps in the offline training process. Therefore, we apply priority experience replay (see Sect. 5.1) and adopt parallel computing to accelerate the convergence of our model. According to the time cost of each step mentioned in Sect. 6.1, the average time spent on offline training is about 4.7 h. This time can be further shortened if a GPU is used, or via reductions to the restart time of a CDB.

6.2.3 Performance improvement

We also evaluate our method on different workloads with CDB-A and show the result in Fig. 12. The tuning performance improvement percentage is shown in Table 8 and

Table 8 Higher throughput and lower latency of CDBTune+ than BestConfig, DBA and OtterTune

Workload	BestConfig		DBA		OtterTune	
	Throughput	Latency	Throughput	Latency	Throughput	Latency
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%

compared with BestConfig, the 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 a state-of-the-art result. Besides, OtterTune performs inferior to the DBA in most cases. This is because we use the trial-and-error samples in RL instead of massive amount of 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 in a high-dimensional space without any past experience. This verifies that the learning-based approach consistently finds better solutions quickly than search-based tuning, and also verifies the superiority of CDBTune+.

CDBTune+ is able to achieve better performance than the other candidates, and especially gains a remarkable improvement under the write-only workload. We show the detailed configurations recommended by OtterTune, BestConfig, DBA, CDB default and CDBTune+ in Tables 11 and 12 in the Appendix. We have following five interesting observations:

Observation 1 We observe that when the buffer pool size is expanded, the configurations which CDBTune+ recommended also increase the size of log file correctly. This indicates that our CDBTune+ can optimize the performance consider the dependencies between knobs rather than each independent knob.

Observation 2 *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 demonstrates that our model can correctly tuning knobs under various workloads, improving both the use of the CPU and the efficiency of the database.

Observation 3 Although tuning the buffer size appropriately is critical to the performance of a cloud database since memory is typically the resource bottleneck [51], we do not tune the *innodb_buffer_pool_size* significantly for achieving higher performance.This demonstrates that our CDBTune+ can tune other knobs (except *innodb_buffer_pool_size*) to enhance database performance while the memory resource is the same.

Observation 4 For those knobs like *innodb_buffer_pool_size*, *binlog_cache_size*, *innodb_log_files_in_group*, *innodb_file_per_table*, *max_binlog_size* and *skip_name_resolve*, we obtain the similar values as the DBA advisors which indicate that the tuning results are explainable to a certain extent and guarantee the overall performance does not differ greatly from the DBAs.

Observation 5 According to the MySQL official manual, 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 in a situation where 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. The practical results verify this method achieves a good effect with the constant reward feedback in RL and this situation occurs 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 (which is an indication that our model learns this rule by itself).

6.2.4 Evaluation on other databases

We evaluate our method on Local MySQL, MongoDB and Postgres where we tune 72 knobs for MongoDB and 169 knobs for Postgres. Figures 13, 14 and 15 show the results. Our method also works well on Local MySQL, MongoDB and Postgres. We find that our method is able to adapt to YCSB workloads using the trained model on the CDB-E database instance on MongoDB, to TPC-C workloads using the trained model on the CDB-D database instance on Postgres ,and to the 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.

6.2.5 Evaluation on different types of storage media

Compared with traditional hard disk drives (HDDs), the emergence of new storage media solid state drives (SSDs) has

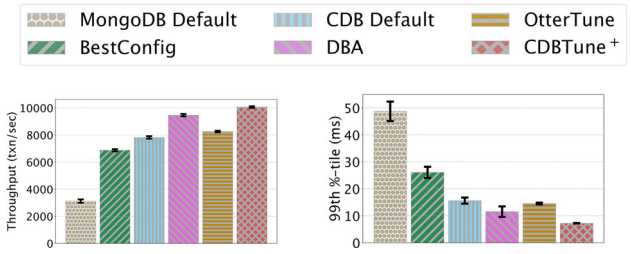


Fig. 13 Performance comparison for the YCSB workload using the instance CDB-E with CDBTune+, MongoDB default, CDB default, BestConfig, DBA and OtterTune (on MongoDB)

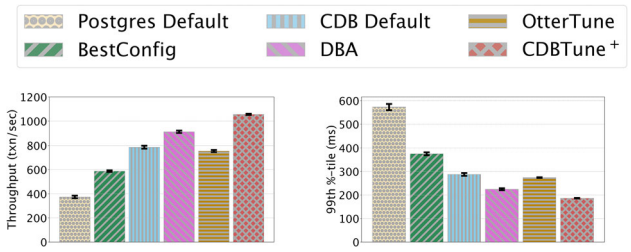


Fig. 14 Performance comparison for the TPC-C workload using the instance CDB-D with CDBTune+, Postgres default, CDB default, BestConfig, DBA and OtterTune (on Postgres)

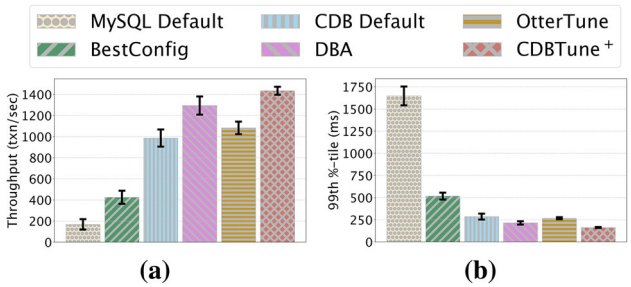


Fig. 15 Performance on TPC-C for local MySQL

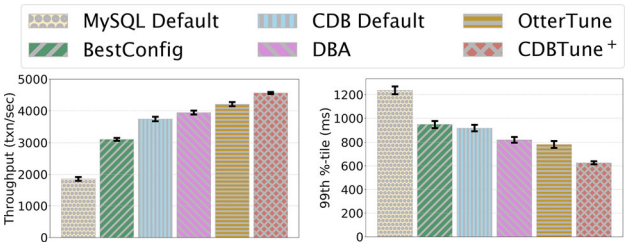


Fig. 16 Performance comparison for TPC-C workload using the SSD-based instance CDB-S-A among CDBTune+, MySQL default, CDB default, BestConfig, DBA and OtterTune

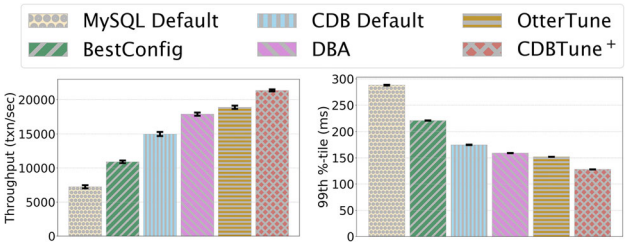


Fig. 17 Performance comparison for TPC-C workload using the NVMe-based instance CDB-N-B among CDBTune+, MySQL default, CDB default, BestConfig, DBA and OtterTune

brought tremendous innovation in I/O performance. In recent years, SSDs are gradually replacing HDDs. In addition, for meeting the growing demand for throughout in storage systems, Non-Volatile Memory Express based SSDs (NVMe SSDs) are the latest development in this domain and deliver unprecedented performance. In order to benefit from the outstanding I/O characteristics of SSDs and NVMe SSDs, SSD and NVMe SSD are widely adopted in the database as a cache layer [17,62]. Therefore, we not only evaluate the performance of our method on the HDDs but also evaluated it on SSDs and NVMe SSDs based databases. Using the database instances CDB-S-A, CDB-N-B and the TPC-C workload, we record the throughput and latency when the model converges. As shown in Figs. 16, 17, CDBTune+ achieves higher performance than other candidates. Note that the results of DBAs are not only worse than the results of CDBTune+ but also worse than that of OtterTune (the DBA achieved better results than OtterTune and worse results than CDBTune+ in the previous experiments using HDDs). We attribute this to the fact that DBAs lack tuning experience with SSDs and NVMe SSDs (less than HDDs) which appear as new media in the database application. In conclusion, we find that CDBTune+ also outperforms other approaches on databases with different storage media.

6.3 Evaluation of reward functions

In this section, we first evaluate the benefits of our custom reward function, then explore how the coefficients C_T and C_L will affect the performance of a DBMS.

6.3.1 Baseline reward functions

For verifying the benefits of our custom reward function in the training and tuning process, we compare it with three baseline reward functions including (1) RF-A: where the performance at the current step is compared only to the performance of the previous step, (2) RF-B: the performance of the current step is compared only to the performance with the initial settings, and (3) RF-C: if the current performance is lower than at the previous step, the corresponding reward part will keep the original method of calculation (for example, its reward remains unchanged even if $\Delta_{t\to t-1}$ is negative in Eq. (8)). We compare the three baseline reward functions to our chosen function RF-CDBTune+ from Sect. 4.2 in terms of the number of iterations until convergence. After multiple steps, if the performance change between two steps does not exceed

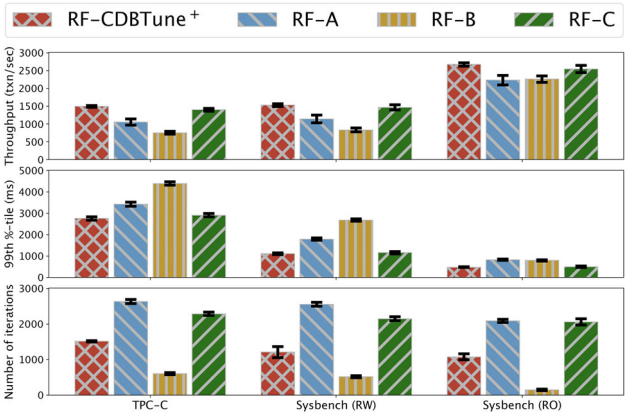


Fig. 18 Number of iterations and performance of CDBTune+ for TPC-C (with CDB-C), Sysbench RW and RO workloads (with CDB-A) respectively using different reward functions. The corresponding number of iterations and performance are collected under the same knob settings

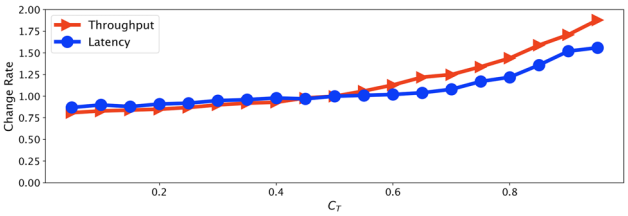


Fig. 19 The coefficient C_T to optimize throughput and latency. Note that $C_T + C_L = 1$

0.5% in five consecutive steps, we consider the model training to have converged. Selecting an appropriate threshold for convergence detection requires a trade-off between training time and model quality. For example, a smaller threshold may give a better result but it will spend more time on model training. We have conducted extensive experiments, and found a convergence threshold of 0.5% to be a sweet spot between convergence time and result quality. Specifically, compared to the performance of the previous step and the initial settings, the reward (corresponding to throughput or latency) calculated by RF-CDBTune+ will be set to 0 if the current performance is lower than that of the previous step but higher than the initial performance.

As shown in Fig. 18, we adopt three different workloads on two different instances CDB-A (8G RAM, 100G Disk) and CDB-C (12G RAM, 200G Disk). In summary, RF-A shows the longest convergence time. What causes this phenomenon is that RF-A just considers the performance at the previous time step, 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 step, causing high 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-CDBTune+, but takes much longer to converge than RF-CDBTune+. If the current performance is lower than that of the previous step, the absolute value part of its reward function is always positive but generally a small number, which will only have a small impact on the system's performance. However, such reward misleads the learning of the intermediate process, causing to a longer convergence time than RF-A. In conclusion, compared with others, our proposed RF-CDBTune+ reword function combines the critical factors comprehensively and achieves the fast convergence speed and best performance.

6.3.2 Varying C_T and C_L

In this section, we investigate our reward function in more detail. In Eq. 9, we present two weights (C_T and C_L) to separately optimize throughput and latency, where C_T+C_L =1. In order to explore how these weights will affect the performance of the DBMS, we compare different setting to a baseline of C_T=C_L=0.5. We change the size of C_T and observe the ratio of throughput to latency compared to our baseline. As shown in Fig. 19, 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 to be larger than that of a smaller C_T (less than 0.5). We observe similar effects for the latency. This is because the changes to C_T and C_L will affect the contributions of throughput and latency to the 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 (latency or throughput sensitivity) to enable the user to trade-off latency and throughput (we set $C_L = 0.6$ and $C_T = 0.4$ in our experiments) according to their specific requirements.

6.4 Adaptability

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

6.4.1 Adaptability to changes in memory size and disk capacity

Compared with local self-built databases, one of the biggest advantages of cloud databases is that data migration or even downtime for reloading is hardly required when resources

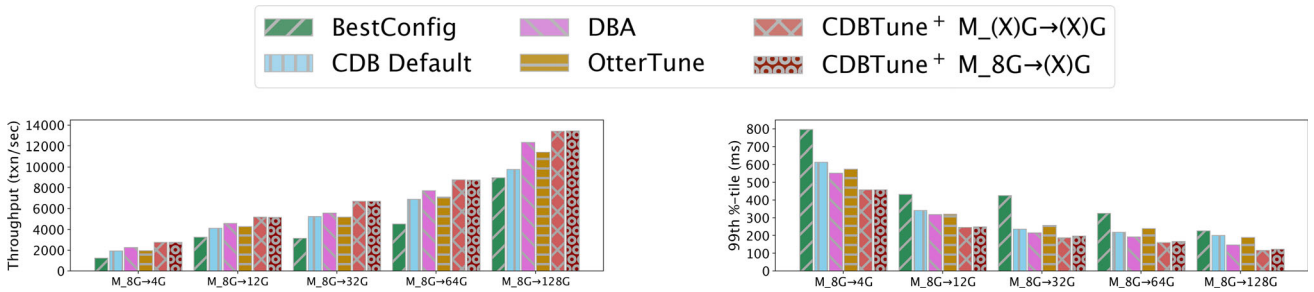


Fig. 20 Performance comparison for Sysbench WO workload when applying the model trained on 8G memory to (X)G memory hardware environment

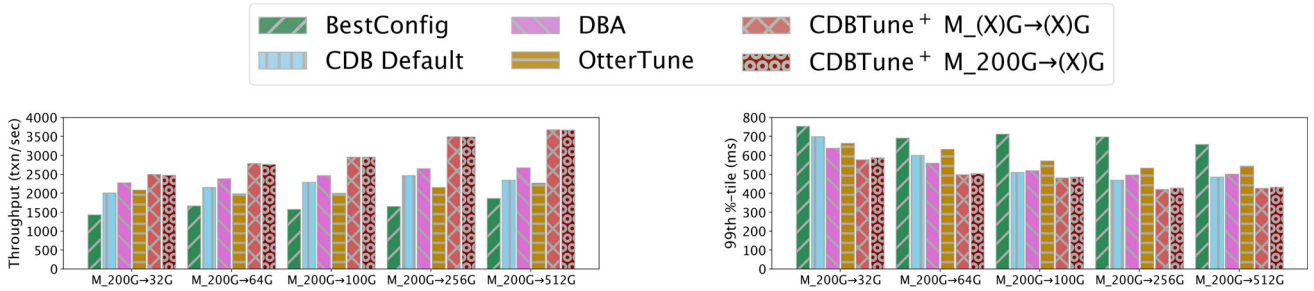


Fig. 21 Performance comparison for Sysbench RO workload when applying the model trained on 200G disk to (X)G disk hardware environment

need to be adjusted. Usually, memory size and disk capacity are what users prefer to adjust. Thus, in the cloud environment, the large number of potential combinations of database memory size and disk capacity prevent us from building a corresponding model for each individual combination. Therefore, a cloud environment naturally requires the DBMS tuning models to adapt well to different configurations. In order to verify that CDBTune+ can adequately optimize the database's performance with different hardware configurations, we experiment on database instances 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 and a disk capacity difference between CDB-C and CDB-X2. For different memory sizes, 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 applying these two configurations. Similarly, for different disk capacities, we apply the same method to complete cross testing and normal testing on CDB-C and CDB-X2. As shown in Figs. 20 and 21, the cross-testing model almost achieves the same performance as the normal-testing model. Moreover, both of the 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 exhibits a high adaptability to a new hardware environments independently how memory size, disk capacity of users change. Note that it would be interesting in future work to explore some monetary cost metrics to give recommendations as to what hardware to use.

6.4.2 Adaptability to workload changes

As mentioned above, we adopt a set of standard testing tools to generate sample data for training in the absence of sufficient historical data. We investigate whether this is sufficient for CDBTune+ to adapt well to different 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 the model called M_RW→TPC-C trained on the read-write workload contained in Sysbench to recommend configurations for the 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 Fig. 22. The tuning performance of the cross-testing model is slightly different from that of the normal-testing model. This finding indicates that our CDBTune+ does not need to establish a new model and adapts well when the workload changes slightly.

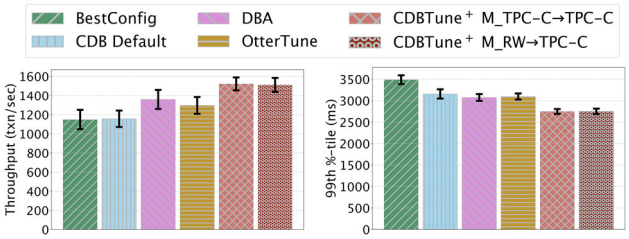


Fig. 22 Performance comparison when applying the model trained on Sysbench RW workloads to TPC-C

6.4.3 Adaptability to cross-storage media

With the rapid development of new storage media, many users will change the storage medium from a HDD to an SSD or NVMe SSD in order to achieve higher performance of the database in a busy I/O scenario. In contrast, for a database instance with low performance requirements, a HDD can be used to replace an SSD or NVMe SSD to reduce costs for users. To verify the adaptability of CDBTune when the storage medium changes, we use database instances CDB-D(HDD), CDB-S-A(SSD) and CDB-N-B(NVMe) in this experiment. Note that there is only a different storage medium between them (same memory size and disk capacity). We first directly utilize the model called M_HDD→{SSD, NVMe} trained on CDB_D to recommend configurations for CDB-S-A or CDB-N-B (cross testing), then use the model called M_HDD→HDD trained on CDB_D to recommend configurations for CDB_D (normal testing), and also finally compare the performance after executing these two configurations. Similar, for different storage media, we use the same method to complete cross testing and normal testing on SSD and NVMe. As shown in Fig. 23, CDBTune$^+$ not only achieves the best performance compared with the other candidates but also makes the cross-testing model almost achieves the same performance as normal-testing model. This demonstrates that when the user changes the storage medium from a HDD to an SSD or NVMe SSD, CDBTune$^+$ will quickly adapt to a new storage medium in the online tuning step. For example, when we changed the storage medium from a HDD to an SSD, we found that CDBTune$^+$ increases the *innodb_log_file_size*. As the number of erase times of the SSD is limited, if a log file is repeatedly erased and written to the same location, than this will result in a large number of I/O delays which may deteriorate database performance. Increasing the value of *innodb_log_file_size* recommended by CDBTune$^+$ can effectively mitigate this situation.

6.4.4 Adaptability to complex setup changes

Users may not only change one of the factors at a time (memory size, disk capacity and workload), but may change multiple factors at the same time, inducing more complicated environmental changes. Therefore, in this section, we will discuss the adaptability of CDBTune$^+$ in this situation. We adopt three different database instances CDB-A (8 GB, 100 GB, HDD), CDB-S-A (16 GB, 200 GB, SSD)

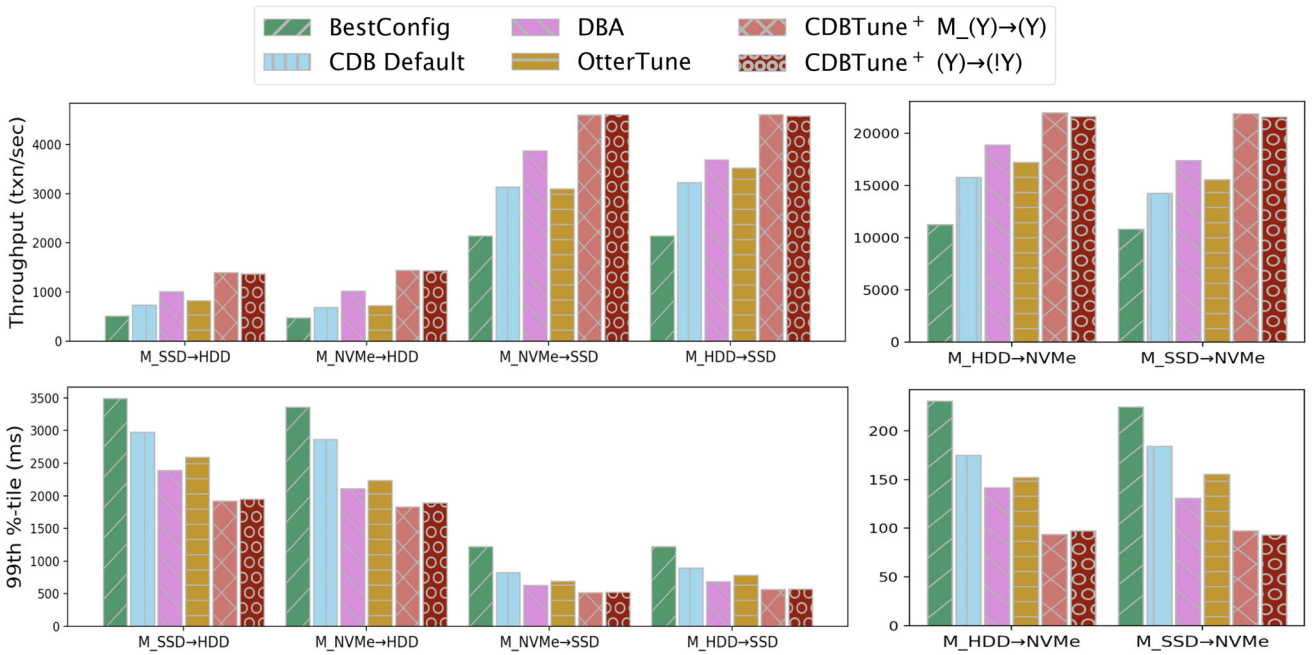


Fig. 23 Performance comparison for the TPC-C workload on cross-storage media. Note that Y is selected from HDD, SSD and NVMe; !Y refers to a medium different from Y

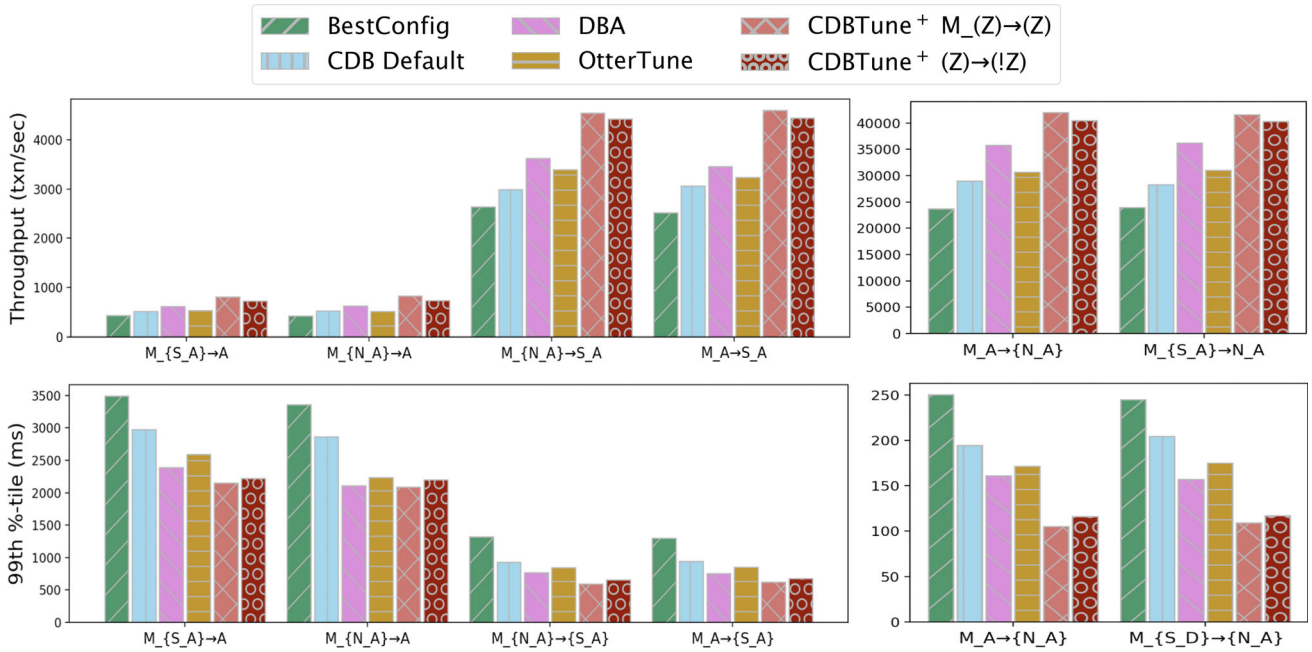


Fig. 24 Performance comparison for the TPC-C workload in light of complex setup changes. Note that Z is selected from CDB-A (A), CDB-S-A (S_A) and CDB-N-A (N_A), and !Z refers to a database instance different from Z

and CDB-N-A (32 GB, 300 GB, NVMe SSD) to conduct our experiments. Note that there are different memory sizes, disk capacities, and storage media between these three database instances. We conducted the normal testing and cross testing experiments and recorded the throughput and latency analogous to the previous experiments. Moreover, to experience more complex environmental changes, we trained our model on the JOB workload in the normal testing to recommend configurations for the TPC-C workload in cross testing. The experimental results are shown in the Fig. 24. For these complex environmental changes, the performance of CDBTune$^+$ in cross testing exhibits a very small decrease compared to the previous three experiments (where only one environmental factor is changed at a time). We attribute this to the fact that CDBTune$^+$ needs a trade off when many environmental factors are changed at the same time.We note that CDBTune$^+$ still achieves the best performance compared to the other candidates.

Summary. The results of the above five presented experiments show that (even with limited training data) our model exhibits strong adaptability to changes in the hardware environment or workload. In comparison, OtterTune relies highly on training datasets, and uses a simple regression approach for recommendation. Thus the performance of the recommended configuration is limited, and OtterTune and BestConfig do not explore the high-dimensional configuration space when the current workload or hardware configuration differs from training condition. Especially in a cloud environment, where we can expect frequent environment changes, the lack of relevant data in the training dataset will result in poor recommendations from OtterTune. Instead, RL enables CDBTune$^+$ simulate human brain, learn towards an optimizing direction, and recommend reasonable knob settings corresponding to the current workload and hardware environment. Thus, CDBTune$^+$ exhibits a high adaptability in cloud environments. In addition, our end-to-ed model is more accurate than the pipelined method (OtterTune) or the heuristic search method (BestConfig), as RL applies an exploration & exploitation strategy, and thereby reduces the possibility of falling into a local optimum. This characteristic results in achieving higher tuning performance but neglects its explainability (similar to the AlphaGo). This is always a tradeoff in many recent ML models. Note that we consider methods to explain the recommendations of our tuning model as an important direction for future work.

7 Related work

Database tuning DBMS tuning has been an interesting and active area of research in the last two decades [1,5,6,10,11,19,25,42,55,60,61,68,69]. 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 [5,9,63,69], and they focused on index optimizations, materialized views and partitions [1–3,7,26,32,41]. Database cracking is a tech-

Table 9 63 metrics include two types: state value and cumulative value

State	Cumulative			
metadata_mem_pool_size	lock_row_lock_current_waits	lock_row_lock_time	buffer_pages_written	os_data_fsyncs
lock_row_lock_time_max	buffer_pool_read_ahead_evicted	lock_row_lock_waits	buffer_pages_read	os_log_fsyncs
lock_row_lock_time_avg	ibuf_merges_discard_delete_mark	buffer_pool_wait_free	buffer_data_reads	lock_deadlocks
buffer_pool_size	innodb_rwlock_s_spin_rounds	buffer_pool_read_ahead	buffer_data_written	lock_timeouts
buffer_pool_pages_total	innodb_rwlock_x_spin_rounds	adaptive_hash_searches	ibuf_merges_insert	log_waits
buffer_pool_pages_misc	innodb_rwlock_s_os_waits	adaptive_hash_searches_btree	ibuf_merges_delete	log_writes
buffer_pool_pages_data	innodb_rwlock_x_os_waits	ibuf_merges_delete_mark	innodb_dblwr_writes	ibuf_merges
buffer_pool_bytes_data	innodb_dblwr_pages_written	ibuf_merges_discard_insert	buffer_pool_reads	ibuf_size
buffer_pool_pages_dirty	innodb_rwlock_s_spin_waits	os_log_pending_fsyncs	buffer_pages_created	dml_reads
buffer_pool_bytes_dirty	innodb_rwlock_x_spin_waits	os_log_pending_writes	log_write_requests	dml_inserts
buffer_pool_pages_free	ibuf_merges_discard_delete	os_log_bytes_written	os_data_reads	dml_deletes
trx_rseg_history_len	buffer_pool_read_requests	innodb_activity_count	os_data_writes	dml_updates
file_num_open_files	buffer_pool_write_requests			
innodb_page_size				

Table 10 64 commonly used knobs (16 most impactful knobs and other extended ones)

16 most impactful knobs	49 extended knobs		
table_open_cache	innodb_adaptive_max_sleep_delay	innodb_autoextend_increment	query_cache_limit
max_connections	innodb_change_buffer_max_size	innodb_buffer_pool_dump_at_shutdown	query_cache_size
innodb_buffer_pool_size	innodb_flush_log_at_timeout	innodb_buffer_pool_load_at_startup	query_cache_type
innodb_buffer_pool_instances	innodb_flushing_avg_loops	innodb_concurrency_tickets	query_prealloc_size
innodb_log_files_in_group	innodb_max_purge_lag	innodb_large_prefix	join_buffer_size
innodb_log_file_size	innodb_old_blocks_pct	innodb_log_buffer_size	tmp_table_size
innodb_purge_threads	innodb_read_ahead_threshold	innodb_max_dirty_pages_pct	max_seeks_for_key
innodb_read_io_threads	innodb_replication_delay	innodb_max_dirty_pages_pct_lwm	innodb_io_capacity
innodb_write_io_threads	innodb_rollback_segments	innodb_random_read_ahead	innodb_lru_scan_depth
innodb_file_per_table	innodb_sync_array_size	eq_range_index_dive_limit	innodb_old_blocks_time
skip_name_resolve	innodb_adaptive_flushing_lwm	innodb_adaptive_hash_index	query_alloc_block_size
binlog_checksum	innodb_sync_spin_loops	max_length_for_sort_data	innodb_purge_batch_size
binlog_cache_size	lock_wait_timeout	read_rnd_buffer_size	innodb_spin_wait_delay
max_binlog_cache_size	metadata_locks_cache_size	table_open_cache_instances	sort_buffer_size
max_binlog_size	innodb_adaptive_flushing	transaction_prealloc_size	thread_cache_size
binlog_format	metadata_locks_hash_instances	binlog_order_commits	max_write_lock_count
	innodb_disable_sort_file_cache		

nique to create indexes adaptively and incrementally as a side-product of query processing [22]. Several studies have proposed different cracking techniques for different aspects including tuple reconstruction [23], updates [19], and convergence [24]. Schuhknecht et al. conducted an experimental study on database cracking to identify the potential, and proposed promising directions in database cracking [44]. 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 [42]. 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 [25]. Sudipto et al. [12] presented the design, implementation, experience, and lessons learned from building the first industrial-strength auto-indexing service for Microsoft Azure SQL Database.

(2) Database configuration tuning Parameter configuration tuning selects appropriate values of parameters (knobs) that can improve a DBMS's performance. Most work in automated database tuning has either focused on specific

parameter tuning (e.g., [46]) or holistic parameter tuning (e.g., [13]).

(i) Specific parameter tuning Techniques for tuning specific classes of parameters include memory management and identifying performance bottlenecks [22,24,44]. IBM DB2 released a self-tuning memory manager that uses heuristics to allocate memory to the DBMS's internal components [46,52]. Tran et al. used linear and quadratic regression models for buffer tuning [53]. A resource monitoring tool has been used with Microsoft's SQL Server for the self-predicting DBMS [38]. Oracle also developed an internal monitoring system to identify bottlenecks due to misconfigurations [14,29]. The DBSherlock tool helps a DBA diagnose problems by comparing slow regions with normal regions based on the DBMS's time-series performance data [64].

(ii) Holistic parameter tuning There are several works for the holistic tuning of 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 [59]. IBM DB2 released a performance Wizard tool for automatically selecting the initial values for the configuration parameters [30]. BerkeleyDB uses influence diagrams to model probabilistic dependencies between configuration knobs, to infer expected outcomes of a particular DBMS configuration [47]. 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 [13]. iTuned is a generic tool that continuously makes minor changes to the DBMS configuration, employing GP regression for automatic configuration tuning [15].

Our system is designed for holistic knob tuning. OtterTune [55] 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 a cloud environment. BestConfig [67] is the closest work that is related to our goals but the techniques applied are completely different. It divides the high-dimensional parameter space into subspaces, and employs search-based methods. However, it does not use 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 databases. The experimental results show that our method achieves much better performance than OtterTune and BestConfig.

Deep Learning for Databases. Deep learning models define a mapping from an input to an output, and learn how to use the hidden layers to produce the corresponding output [31]. Although deep learning has successfully been applied to solving computationally intensive learning tasks in many domains [18,20,21,28,57], there are few studies that have used deep learning techniques to solve database tuning problems [58]. Reinforcement learning is able to discover the best action through the trial-and-error method by either exploiting current knowledge or exploring unknown states to maximize a cumulative reward [34,48,49].

Recently, several researches utilized deep learning or RL model to solve problems in database research. Tzoumas et al. [54] 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. [4] used an RL model for adaptive performance tuning of database applications. Pavlo et al. [40] presented the architecture of Peloton for workload forecasting and action deployment under the algorithmic advancements in deep learning. Marcus et al. [35] used deep RL to determine join orders. Ortiz et al. [39] 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. [45] used the deep RL model to automatically administer a DBMS by defining a problem environment. Tan et al. [50] proposed iBTune based on building a pairwise deep neural network to tune DBMS buffer pool sizes by using a large deviation analysis for LRU caching models. Their experiment shows that iBTune can save more than 17% of memory resources compared to the original system that only relies on experienced DBAs.

Table 11 Detailed configurations recommended by OtterTune for the same workloads in Table 12, while knobs are not identical

Knobs	RO	WO	RW
innodb_buffer_pool_size	6.3G	7.2G	7.3G
innodb_buffer_pool_instances	15	6	4
innodb_log_file_size	1.8G	2.9G	2.2G
innodb_adaptive_flushing_lwm	13	19	23
innodb_max_dirty_pages_pct	19	28	7
innodb_large_prefix	OFF	OFF	OFF
query_cache_size	87M	63M	34M
innodb_io_capacity	24317	37467	17802
thread_cache_size	18	12	26
innodb_log_buffer_size	13M	21M	8M
metadata_locks_cache_size	2041	1532	647
table_open_cache_instances	8	39	38
innodb_disable_sort_file_cache	OFF	OFF	OFF
join_buffer_size	5M	6M	6M
innodb_sync_array_size	3	6	4
sort_buffer_size	35M	31M	87M

Table 12 Detailed configurations recommended by BestConfig, DBA and CDBTune for three workloads of Sysbench: RO, WO and RW, while CDB default is the default configuration setting provided by CDB for users

Knobs	CDB default	BestConfig			DBA			CDBTune		
		RO	WO	RW	RO	WO	RW	RO	WO	RW
innodb_buffer_pool_size	7.2G	2.4G	6.6G	1.8G	7.4G	7.6G	7.1G	7.2G	7.9G	7.8G
max_connections	1600	34457	87931	13911	8500	10240	20000	1100	1100	36780
table_open_cache	512	171438	339194	355079	65536	51200	20480	365124	58992	240075
skip_name_resolve	ON	OFF	OFF	OFF	ON	OFF	OFF	ON	OFF	OFF
innodb_buffer_pool_instances	8	47	1	52	8	4	1	24	15	6
innodb_log_files_in_group	2	31	78	10	4	5	4	3	3	2
innodb_log_file_size	0.5G	1G	40M	0.9G	4G	4G	2G	5.6G	6.2G	4.7G
innodb_purge_threads	1	11	1	20	4	8	2	2	16	8
innodb_read_io_threads	12	18	60	1	8	6	6	68	3	15
innodb_write_io_threads	12	18	60	1	8	10	2	8	29	13
innodb_file_per_table	ON	ON	OFF	OFF	ON	ON	ON	ON	ON	ON
binlog_checksum	NONE	NONE	CRC32	CRC32	NONE	CRC32	NONE	CRC32	CRC32	NONE
binlog_cache_size	32M	1.1G	0.2G	0.5G	64M	200M	200M	89M	427M	139M
max_binlog_cache_size	7.8G	2.5G	1.9G	0.7G	2G	3G	2G	2.3G	4.4G	3.7G
max_binlog_size	1G	0.3G	0.8G	0.2G	0.5G	0.9G	0.5G	0.6G	1G	0.5G
binlog_format	MIXED	ROW	MIXED	MIXED	ROW	ROW	ROW	ROW	MIXED	ROW

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.

8 Conclusion

In this paper, we proposed an end-to-end automatic DBMS configuration tuning system CDBTune$^+$ that can recommend well-working knob settings in complex cloud environments. CDBTune$^+$ applies exploration RL to learn the best settings with limited samples. Moreover, our custom reward function can effectively improve the tuning efficiency in high-dimensional continuous space for the knobs. Extensive experimental results showed that CDBTune$^+$ produced configurations for various workloads that greatly improved performance with higher throughput and lower latency compared to the state-of-the-art tuning tools and DBA experts. We also demonstrated that CDBTune$^+$ adapts well to changes in the operating environment. In future work, we intend to explore other ML solutions (transfer learning, meta learning, genetic algorithms and etc.) to improve the database tuning performance further.

Acknowledgements 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 (61632016, 61472198), 973 Program of China No. 201-5CB358700.

Appendix

A introduction of CDB

CDB is redeveloped based on MySQL (now support MongoDB and Postgres) by Tencent. It is a database hosting service that combines high performance, high availability, high security, high scalability and ease of use. Users can easily deploy, use and expand CDB (whose kernel is MySQL, MongoDB or Postgres) within a few minutes in the cloud. Moreover, the size of hardware capacity is allowed to be adjusted flexibly without downtime. CDB for MySQL provides users with a full set of database maintenance solutions such as backup, monitoring, rapid expansion, data transmission and so on in order to simply IT operations for users and focus more on the business development. At present, CDB owns more than 200,000 users and 5,000,000 instances from 50 countries around the world.

B metrics

This section lists 63 internal metrics which represent the current state of a database. They are classified into two types: state value and cumulative value, as shown in Table 9.

C knobs

CDBTune$^+$ exposes 65 commonly used knobs for users to tune the performance of their cloud database. As shown in Table 10, there are 16 most impactful knobs selected by DBA and other expended 49 ones are used in the experiments.

D knob recommendations

This section shows the DBMS configurations of CDB default, and those generated by BestConfig, the DBA, our CDBTune and OtterTune in Tables 12 and 11 for three workloads of Sysbench: RO, WO, and RW running on CDB for MySQL. Table 12 displays the 16 most frequently used knobs in CDB. Table 11 displays the 16 most impactful knobs for OtterTune. For instance, the table_open_cache in common knobs stands for the number of open tables for all threads and the query_cache_size stands for the amount of memory allocated for caching query results.

References

1. Agrawal, S., Bruno, N., Chaudhuri, S., et al.: Autoadmin: Self-tuning database systemstechnology. IEEE Data Eng. Bull. **29**(3), 7–15 (2006)
2. Agrawal, S., Chaudhuri, S., Kollar, L., Marathe, A., Narasayya, V., Syamala, M.: Database tuning advisor for microsoft sql server 2005. In: ACM SIGMOD, pp. 930–932. ACM, (2005)
3. Agrawal, S., Narasayya, V., Yang, B.: Integrating vertical and horizontal partitioning into automated physical database design. In: ACM SIGMOD, pp. 359–370. ACM, (2004)

4. Basu, D., Lin, Q., Vo, H.T., Vo, H.T., Yuan, Z., Senellart, P.: Regularized Cost-Model Oblivious Database Tuning with Reinforcement Learning. Springer, Berlin Heidelberg (2016)
5. Belknap, P., Dageville, B., Dias, K., Yagoub, K.: Self-tuning for sql performance in oracle database 11g. In: ICDE, pp. 1694–1700. IEEE, (2009)
6. Bernstein, P., et al.: The asilomar report on database research. ACM Sigmod record **27**(4), 74–80 (1998)
7. Bruno, N., Chaudhuri, S.: Automatic physical database tuning: a relaxation-based approach. In: ACM SIGMOD, pp. 227–238. ACM, (2005)
8. Cao, W., Liu, Y. et al: POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database. In: 18th USENIX Conference on File and Storage Technologies (FAST 20), pp. 29–41, Santa Clara, CA, February (2020). USENIX Association
9. Chaudhuri, S., Narasayya, V.: Autoadmin "what-if" index analysis utility. In: ACM SIGMOD, pp. 367–378, (1998)
10. Chaudhuri, S., Narasayya, V.: Self-tuning database systems: A decade of progress. In: VLDB, pp. 3–14, (2007)
11. Chaudhuri, S., Weikum, G.: Rethinking database system architecture: Towards a self-tuning risc-style database system. In: VLDB, pp. 1–10, (2000)
12. Das, S., Grbic, M. et al.: Automatically indexing millions of databases in microsoft azure sql database. In: Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19, page 666–679, New York, NY, USA, (2019). Association for Computing Machinery
13. Debnath B.K., Lilja, D.J., Mokbel, M.F.: Sard: A statistical approach for ranking database tuning parameters. In: ICDEW, pp. 11–18. IEEE, (2008)
14. Dias, K., Ramacher, M., Shaft, U., Venkataramani, V., Wood, G.: Automatic performance diagnosis and tuning in oracle. In: CIDR, pp. 84–94, (2005)
15. Duan, S., Thummala, V., Babu, S.: Tuning database configuration parameters with ituned. VLDB Endowment **2**(1), 1246–1257 (2009)
16. Dundar, M., Krishnapuram, B. et al.: Learning classifiers when the training data is not iid. In: Proceedings of the 20th International Joint Conference on Artifical Intelligence, IJCAI'07, page 756–761, San Francisco, CA, USA, (2007). Morgan Kaufmann Publishers Inc
17. Ghodsnia, P.: Effective use of ssds in database systems, (2018)
18. Goldberg, Y.: A primer on neural network models for natural language processing. Computer Science, (2015)
19. Graefe, G., Kuno, H.A.: Self-selecting, self-tuning, incrementally optimized indexes. In: EDBT, pp. 371–381, (2010)
20. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: CVPR, pp. 770–778, (2016)
21. Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. Science **313**(5786), 504–507 (2006)
22. Idreos, S., Kersten, M.L., Manegold, Stefan: Database cracking. In: CIDR, pp. 68–78, (2007)
23. Idreos, S.K., Martin L., Manegold, S.: Self-organizing tuple reconstruction in column-stores. In: ACM SIGMOD, pp. 297–308, (2009)
24. Idreos, S., Manegold, S., Kuno, H.A., Graefe, G.: Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. PVLDB **4**(9), 585–597 (2011)
25. Idreos, S., Zoumpatianos, K. et al.: The data calculator: Data structure design and cost synthesis from first principles and learned cost models. In: ACM SIGMOD, pp. 535–550, (2018)
26. Ilyas, I.F., Markl, V., Haas, P., Brown, P., Aboulnaga, A.: Cords: automatic discovery of correlations and soft functional dependencies. In: ACM SIGMOD, pp. 647–658. ACM, (2004)
27. Kaelbling, L.P., et al.: Reinforcement learning: A survey. J. Artif. Int. Res., 4(1):237–285, May (1996)
28. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: NIPS, pp. 1097–1105, (2012)
29. Kumar, S.: Oracle database 10g: The self-managing database, (2003)
30. Kwan, E., Lightstone, S. et al.: Automatic configuration for ibm db2 universal database. In: Proc. of IBM Perf Technical Report, (2002)
31. Lecun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436 (2015)
32. Lightstone, S.S., Bhattacharjee, B.: Automated design of multidimensional clustering tables for relational databases. In: VLDB, pp. 1170–1181, (2004)
33. Lillicrap, T.P., Hunt, J.J., Pritzel, A. et al.: Continuous control with deep reinforcement learning. arXiv:1509.02971 (2015)
34. Maglogiannis, V., Naudts, D., Shahid, A., Moerman, I.: A q-learning scheme for fair coexistence between LTE and wi-fi in unlicensed spectrum. IEEE Access **6**, 27278–27293 (2018)
35. Marcus, R., Papaemmanouil, O.: Deep reinforcement learning for join order enumeration. arXiv:1803.00055 (2018)
36. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.A.: Playing atari with deep reinforcement learning. arXiv:1312.5602 (2013)
37. Mohan, C., Haderle, D., et al.: Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. Acm Trans Database Syst **17**(1), 94–162 (1992)
38. Narayanan, D., Thereska, E., Ailamaki, A.: Continuous resource monitoring for self-predicting dbms. In: null, pp. 239–248. IEEE, (2005)
39. Ortiz, J., Balazinska, M., Gehrke, J., Keerthi, S.S.: Learning state representations for query optimization with deep reinforcement learning. arXiv:1803.08604 (2018)
40. Pavlo, A., Angulo, G., Arulraj, J., Lin, H., Lin, J., Ma, L., Menon, P., Mowry, T.C., Perron, M., Quah, I. et al.: Self-driving database management systems. In: CIDR, (2017)
41. Rao, J., Zhang, C. et al.: Automating physical database design in a parallel database. In: ACM SIGMOD, pp. 558–569. ACM, (2002)
42. Richter, S., Quiané-Ruiz, J.-A., et al.: Towards zero-overhead static and adaptive indexing in hadoop. VLDB J. **23**(3), 469–494 (2014)
43. Schaul, T., Quan, J., et al.: Prioritized experience replay. Computer Science, (2015)
44. Schuhknecht, F.M., Jindal, A., Dittrich, J.: The uncracked pieces in database cracking. PVLDB **7**(2), 97–108 (2013)
45. Sharma, A., Schuhknecht, F.M., Dittrich, J.: The case for automatic database administration using deep reinforcement learning. (2018)
46. Storm, A.J., Garcia-Arellano, C., Lightstone, S.S., Diao, Y., Surendra, M.: Adaptive self-tuning memory in db2. In: VLDB, pp. 1081–1092. VLDB, (2006)
47. Sullivan, D.G., Seltzer, M.I., Pfeffer, A.: Using probabilistic reasoning to automate software tuning, vol. 32. ACM, (2004)
48. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction, bradford book. IEEE Transactions on Neural Networks **16**(1), 285–286 (2005)
49. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. (2011)
50. Tan, J., Zhang, T., et al.: Ibtune: Individualized buffer tuning for large-scale cloud databases. Proc. VLDB Endow. **12**(10), 1221–1234 (2019)
51. Tan, J., Zhang, T., Li, F., Chen, J., Zheng, Q., Zhang, P., Qiao, H., Shi, Y., Cao, W., Zhang, R.: ibtune: Individualized buffer tuning for large-scale cloud databases. Proceedings of the VLDB Endowment **12**(10), 1221–1234 (2019)

52. Tian, W., Martin, P., Powley, W.: Techniques for automatically sizing multiple buffer pools in db2. In: Centre for Advanced Studies on Collaborative research, pp. 294–302. IBM Press, (2003)
53. Tran, D.N., Huynh, P.C., Tay, Y.C., Tung, A.K.H.: A new approach to dynamic self-tuning of database buffers. TOS **4**(1), 3 (2008)
54. Tzoumas, K., Sellis, T., Jensen, C.S.: A reinforcement learning approach for adaptive query processing. History (2008)
55. Van Aken, D., Pavlo, A., Gordon, G.J., Zhang, B.: Automatic database management system tuning through large-scale machine learning. In: ACM SIGMOD, pp. 1009–1024, (2017)
56. Verbitski, A., Gupta, A., Saha, D., et al.: Amazon aurora: Design considerations for high throughput cloud-native relational databases. In: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD '17, page 1041–1052, (2017)
57. Wang, L., Ye, J., Zhao, Yiyang, W., Wei, L., Ang, S., Shuaiwen L., Xu, Z., Kraska, T.: Superneurons: Dynamic gpu memory management for training deep neural networks. (2018)
58. Wang, W., Zhang, M., Chen, G., Jagadish, H.V., Ooi, B.C., Tan, K.-L.: Database meets deep learning: Challenges and opportunities. ACM SIGMOD Record **45**(2), 17–22 (2016)
59. Weikum, G., Hasse, C., Mönkeberg, A., Zabback, P.: The comfort automatic tuning project. Information systems **19**(5), 381–432 (1994)
60. Weikum, G., Moenkeberg, A., Hasse, C., Zabback, P.: Self-tuning database technology and information services: from wishful thinking to viable engineering. In: VLDB, pp. 20–31. Elsevier, (2002)
61. Wiese, D., Rabinovitch, G., Reichert, M., Arenswald, S.: Autonomic tuning expert: a framework for best-practice oriented autonomic database tuning. In: Center for advanced studies on collaborative research, page 3. ACM, (2008)
62. Xu, Q., Siyamwala, H., et al.: Performance analysis of nvme ssds and their implication on real world databases. In: SYSTOR, pp. 6:1–6:11, (2015)
63. Yagoub, K., Belknap, P., Dageville, B., et al.: Oracle's sql performance analyzer. IEEE Data Eng. Bull. **31**(1), 51–58 (2008)
64. Yoon, D.Y., et al.: Dbsherlock: A performance diagnostic tool for transactional databases. In: ACM SIGMOD, pp. 1599–1614. ACM, (2016)
65. Zhan, C., Su, Ma., et al.: Analyticdb: Real-time olap database system at alibaba cloud. 12(12):2059–2070, August (2019)
66. Zhang, H., Wang, J., et al.: Learning to design games: Strategic environments in reinforcement learning. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18, pp. 3068–3074, 7 (2018)
67. Zhu, Y., Liu, J., Guo, M., Bao, Y., et al.: Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In: SoCC, pp. 338–350. ACM, (2017)
68. Zilio, D.C.: Physical database design decision algorithms and concurrent reorganization for parallel database systems. (1998)
69. Zilio, D.C., Rao, J., Lightstone, S., Lohman, G., Storm, A., Garcia-Arellano, C., Fadden, S.: Db2 design advisor: integrated automatic physical database design. In: VLDB, pp. 1087–1097, (2004)