QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning

Guoliang Li[†], Xuanhe Zhou[†], Shifu Li[‡], Bo Gao[‡]
[†] Department of Computer Science,Tsinghua University, Beijing, China [‡] Huawei Company
liguoliang@tsinghua.edu.cn, zhouxuanhe@bupt.edu.cn, {gaobo15,lishifu}@huawei.com

ABSTRACT

Database knob tuning is important to achieve high performance (e.g., high throughput and low latency). However, knob tuning is an NP-hard problem and existing methods have several limitations. First, DBAs cannot tune a lot of database instances on different environments (e.g., different database vendors). Second, traditional machine-learning methods either cannot find good configurations or rely on a lot of high-quality training examples which are rather hard to obtain. Third, they only support coarse-grained tuning (e.g., workload-level tuning) but cannot provide fine-grained tuning (e.g., query-level tuning).

To address these problems, we propose a query-aware database tuning system `QTune` with a deep reinforcement learning (DRL) model, which can efficiently and effectively tune the database configurations. `QTune` first featurizes the SQL queries by considering rich features of the SQL queries. Then `QTune` feeds the query features into the DRL model to choose suitable configurations. We propose a Double-State Deep Deterministic Policy Gradient (DS-DDPG) model to enable query-aware database configuration tuning, which utilizes the actor-critic networks to tune the database configurations based on both the query vector and database states. `QTune` provides three database tuning granularities: query-level, workload-level, and cluster-level tuning. We deployed our techniques onto three real database systems, and experimental results show that `QTune` achieves high performance and outperforms the state-of-the-art tuning methods.

PVLDB Reference Format:
Guoliang Li, Xuanhe Zhou, Shifu Li, Bo Gao. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *PVLDB*, 12(12): 2118 - 2130, 2019.
DOI: https://doi.org/10.14778/3352063.3352129

1. INTRODUCTION

Databases have hundreds of knobs (or parameters) and most of knobs are in continuous space. For example, MySQL, PostgreSQL, and MongoDB have 215, 247, 132 knobs respectively. Database knob tuning is important to achieve high performance (e.g., high throughput and low latency) [2, 5, 34]. Traditionally, databases rely on DBAs to tune the knobs. However, this traditional method has several limitations. First, knob tuning is an NP-hard problem [27] and DBAs can only tune a small percentage of the knobs and may not find a good global knob configuration. Second, DBAs require to spend a lot of time (e.g., several days) to tune the database, and thus they are not efficient to tune many database instances under different environments (e.g., cloud databases). Third, DBAs are usually good at tuning a specific database, e.g., MySQL, but cannot tune other databases, e.g., PostgreSQL. These limitations are extremely severe for tuning cloud databases, because they have to tune a lot of database instances on different environments (e.g., different CPU, RAM and disk).

Recently, there are some studies on automatic knob tuning, e.g., BestConfig [38], OtterTune [2], and CDBTune [36]. However, BestConfig uses a heuristic method to search for the optimal configuration from the history and may not find good knob values if there is no similar configuration in the history. OtterTune utilizes machine-learning techniques to collect, process and analyze knobs and tunes the database by learning DBAs' experiences from the historical data. However, OtterTune relies on a large number of high-quality training examples from DBAs' experience data, which are rather hard to obtain. CDBTune uses deep reinforcement learning (DRL) to tune the database by using a try-and-error strategy. However, CDBTune has three limitations. First, CDBTune requires to run a SQL query workload multiple times in the database to get an appropriate configuration, which is rather time consuming. Second, CDBTune only provides a coarse-grained tuning (i.e., tuning for read-only workload, read-write workload, write-only workload), but cannot provide a fine-grained tuning (i.e., tuning for a specific query workload). Third, it directly uses the existing DRL model, which assumes that the environment can only be affected by reconfiguring actions, but cannot utilize the query information, which is more important for configuration tuning and environment updates.

To address these problems, we propose a query-aware database tuning system `QTune` using a DRL model, which can efficiently and effectively tune the databases. `QTune` first featurizes the SQL queries by considering rich features of the SQL queries, including query type, tables, and query cost. Then `QTune` feeds the query features into the DRL model to dynamically choose suitable configurations. Different from the traditional DRL methods [16, 30], we propose a Double-State Deep Deterministic Policy Gradient

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352129

(DS-DDPG) model using the actor-critic networks. The DS-DDPG model can automatically solve the tuning problem by learning the actor-critic policy according to both the database states and query information. Moreover, `QTune` provides three database tuning granularities. The first is query-level tuning, which finds a good configuration for each SQL query. This method can achieve low latency but low throughput, because it cannot run the SQL queries in parallel. The second is workload-level tuning, which finds a good configuration for a query workload. This method can achieve high throughput but high latency, because it cannot find a good configuration for every SQL query. The third is cluster-level tuning, which clusters the queries into several groups and finds a good database configuration for the queries in each group. This method can achieve both high throughput and low latency, because it can find the good configuration for a group of queries and run the queries in each group in parallel. Thus `QTune` can make a trade-off between latency and throughput based on a given requirement, and provide both coarse-grained tuning and fine-grained tuning. We propose a deep learning based query clustering method to classify queries according to the similarity of their suitable configurations.

We make the following contributions in this paper.

(1) We propose a query-aware database tuning system using deep reinforcement learning, which provides three database tuning granularities (see Section 2).

(2) We propose a SQL query featurization model that featurizes a SQL query to a vector by using rich SQL features (see Section 3).

(3) We propose the DS-DDPG model, which embeds the query features and utilizes the actor-critic algorithm to learn the relations among queries, database state and configurations to tune database configurations (see Section 4).

(4) We propose a deep learning based query clustering method to classify queries according to the similarity of their suitable configurations (see Section 5).

(5) We conducted extensive experiments on various query workloads and databases. Experimental results showed that `QTune` achieved high performance and outperformed the state-of-the-art tuning methods (see Section 6).

2. SYSTEM OVERVIEW

In this section, we present the system overview of `QTune`. `QTune` supports three types of tuning requests based on different tuning granularities.

Query-level Tuning. For each query, it first tunes the database knobs and then executes the query. Note that the session-level knobs (e.g., bulk write size) can be concurrently tuned for different queries, while the system-level knobs (e.g., working memory size) cannot be concurrently tuned because when we tune these knobs for a query, the system cannot process other queries. This method can optimize the latency but may not achieve high throughput.

Workload-level Tuning. It tunes the database knobs for the whole query workload. This method cannot optimize the query latency, because different queries may require to use different best knob values. This method, however, can achieve high throughput, because different queries can be concurrently processed after setting the newly tuned knobs.

Cluster-level Tuning. It partitions the queries into different groups such that the queries in the same group should use the same tuning knob values while the queries in different groups should use different knob values. Next it tunes the knobs for each query group and executes the queries in each group in parallel. This method can optimize both the latency and throughput.

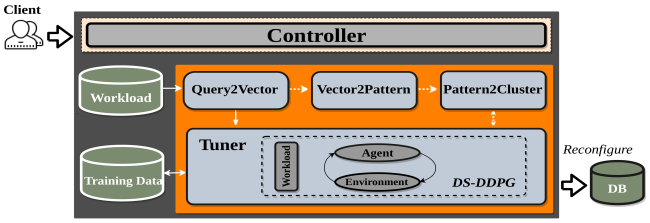


Figure 1: The QTune Architecture

Architecture. Figure 1 shows the architecture of `QTune`, which contains five main components. Figure 2 shows the workflow. The client interacts with `Controller` to pose tuning requests. `Query2Vector` featurizes each query into a vector. It first analyzes the SQL query, extracts the query plan and the estimated cost of each query from the database engine, and uses this information to generate a vector. Based on the feature vectors, `Tuner` recommends appropriate knobs and then the database executes these queries based on the new knob values. `Tuner` uses the deep reinforcement model DS-DDPG to tune the model and recommend continuous knob values as a new configuration. `Tuner` also requires to train the model using some training data, which is stored in the `Training Data` repository. We will explain more details of `Tuner` in Section 4.

For query-level tuning, `Query2Vector` generates a feature vector for the given query. `Tuner` takes this vector as input, and recommends continuous knob values. Next the system executes the query based on the recommended knob values.

For workload-level tuning, `Query2Vector` generates a feature vector for each query in the workload and merges them to generate a unified vector. `Tuner` takes this unified vector as input, and recommends knob values. Next the system executes the queries based on the recommended knob values.

For cluster-level tuning, `Query2Vector` first generates a feature vector for each query and `Tuner` learns a configuration pattern for each query which can learn the *continuous* knob values that best match the query. However, `Tuner` may be expensive to generate the configuration pattern for all the queries. To improve the performance, we propose a deep learning model, `Vector2Pattern`, which learns a *discrete* value for the knobs. Then `Pattern2Cluster` classifies these queries based on their discrete configuration patterns. Note that `Vector2Pattern` uses deep learning to predict the configuration pattern for each query, which also involves a training step that learns a discrete configuration pattern for a given query. For example, considering the knob *host_cache_size* in MySQL, which limits the size of the host cache, `Tuner` recommends a continuous value between 0 to 65536, while `Vector2Pattern` recommends a discrete value in {-1, 0, +1}. After clustering, we get a set of groups. For each query group, `Tuner` recommends appropriate configurations and then the database executes these queries in the group based on the new knob values. So in Figure 2, we provide two cluster-level algorithms. Cluster-level(C) uses the DRL model to learn continuous values while Cluster-level(D) uses the DL model to learn discrete values.

3. QUERY FEATURIZATION

In this section we introduce how to vectorize the queries. There are several challenges in query featurization. The

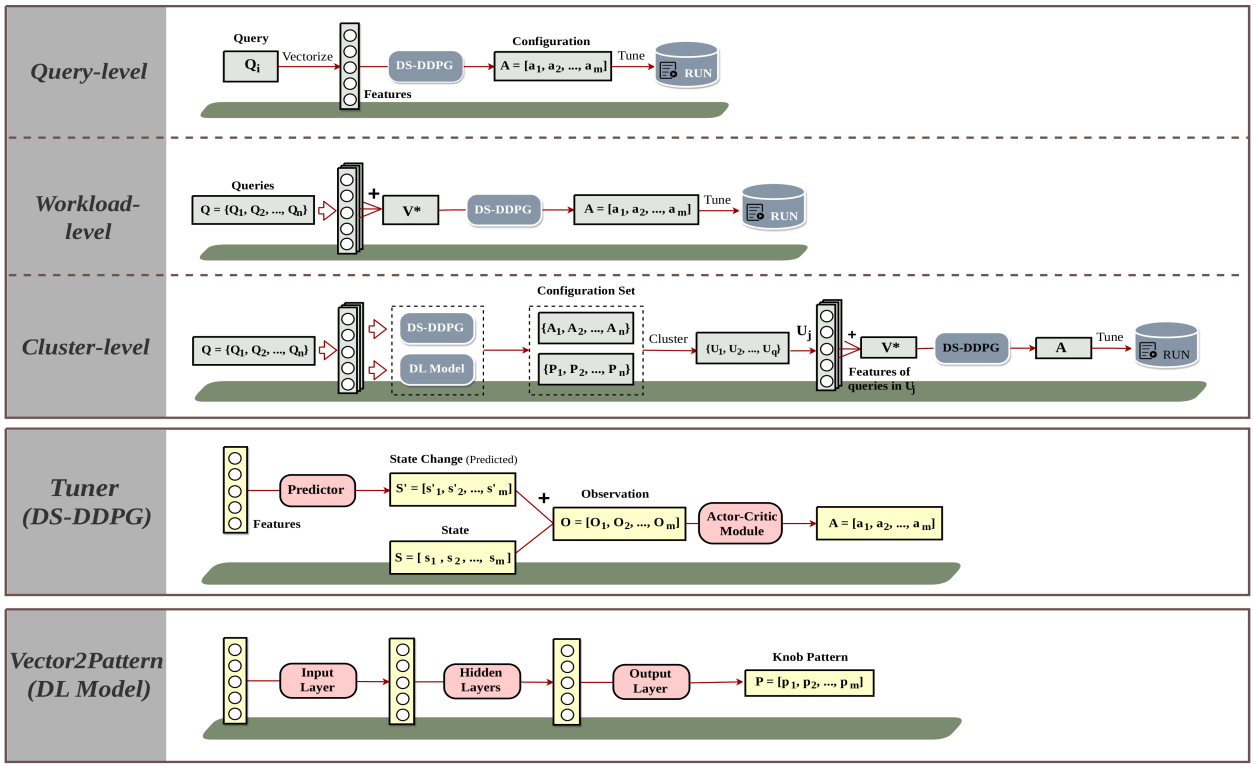


Figure 2: Workflow of QTune

first is to capture the query information, e.g., which tables are involved in the query. The second is to capture the query cost of processing the query, e.g., the selection cost and join cost. The third is to uniformly featurize the query and cost information such that each feature of the vector for different queries has the same meaning. Next we discuss how to address these challenges in the following sections.

3.1 Query Information

A SQL query includes query type (e.g., insert, delete, select, update), tables, attributes, operations (e.g., selection, join, groupby). Query type is important as different query types have different query cost (e.g, OLTP and OLAP have different effect on the database), and thus we need to capture the query type information in the vector. Tables involved in a query are also important, because the data volumes and structures of tables will significantly affect the database performance. Based on the table information, our tuning model decides whether the current system configuration can provide high performance; if not, our tuning system can tune the corresponding knobs. For example, if the buffer is not large enough, we can increase the buffer.

Note that we do not featurize the attributes (i.e., columns) and operations (i.e., selection conditions) due to three reasons. First, the query cost will capture the operation information and cost, and we do not need to maintain duplicated information. Second, operations are too specific and adding specific operations into the vectors will reduce the generalization ability. Third, the attributes and operations will be frequently updated and it requires to redesign the model for the updates. We will compare with the method that also considers attributes and operations in Section 6.1.2 .

In summary, for query information, we maintain a $4 + |T|$ dimensional vector, where $|T|$ is the number of tables in the database. The first four features capture the query types, e.g., insert, select, update, delete. For an insert/select/update/delete query, the corresponding value is 1; 0 otherwise. Each last $|T|$ feature denotes a table. If the query contains the table, the corresponding value is 1; 0 otherwise. For example, Figure 3 shows a query vector. There are 8 tables. The first 12 features are used for query information. It is a selection query and uses tbl1, tbl2 and tbl3, so the first four values are 1 and the other 8 values are 0.

3.2 Cost Information

The cost information captures the cost of processing the query. However, a query usually has many possible physical plans and each plan has different query cost. So it is not realistic to directly parse the query statement to extract query cost. Instead, we utilize the query plan generated by the query optimizer, which has a cost estimation for each operation. Figure 3 shows an example query plan, where each node has a cost estimation. As each database has a fixed number of operations, e.g., scan, hash join, aggregate, we use the cost on these operations to capture the cost information. For example, in PostgreSQL there are 38 operations. Note that an operation may appear in different nodes of the tree plan, and the cost of the same operation should be summed up as the corresponding cost value in the query cost vector. For example, in Figure 3, the value of hash join equals to the sum of the costs in the two Hash Join nodes. After gaining the query cost, we normalize the cost by subtracting mean and dividing the std deviation.

In summary, for cost information, we maintain a $|P|$ dimensional vector, where P is the set of operations in database, and $|P|$ is the number of operations.

3.3 Character Encoding

We concatenate the query vector and cost vector to generate an overall vector of a query. For example, Figure 3 shows the vector of a SQL query.

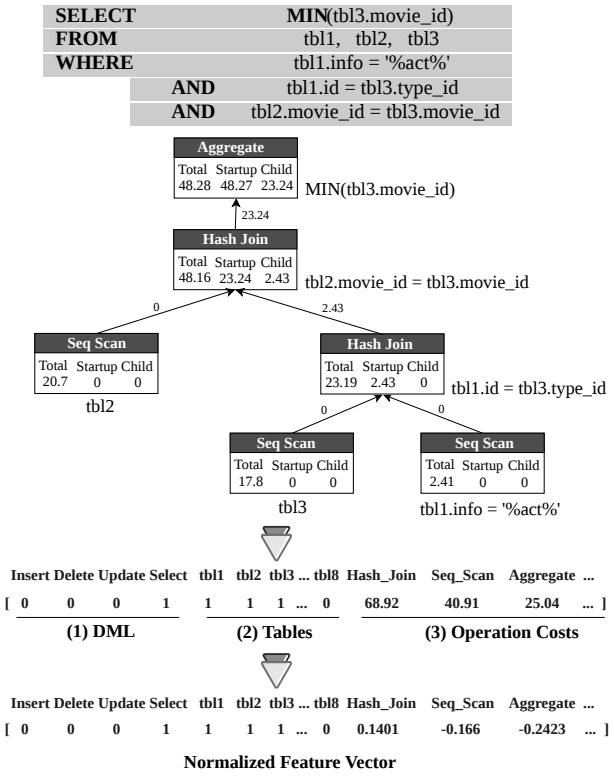


Figure 3: Character Encoding.

Vector for multiple queries. Given multiple queries $q_1, q_2, \cdots, q_m$, suppose their vectors are $v_1, v_2, \cdots, v_m$ respectively. To tune the database for this query workload, we need to combine the vectors together. To this end, for each query vector, we need to consider all the query types and tables, and thus we compute the union of the query vectors. And for each table, if the value is 1, we replace it with the row number of the table. Thus it can capture the actions like deleting/inserting rows and improve system's adaptivity; for cost vector, we need to sum up all the costs. Thus we can combine the vector as follows.

$$[\cup_1^m v_i[1], \ldots, \cup_1^m v_i[4+|T|], \sum_1^m v_i[5+|T|], \cdots, \sum_1^m v_i[4+|T|+|P|]]$$

Supporting Update. We discuss how to support the update of the databases. The database update can only affect the query vector, as the cost vector is computed on-the-fly from the optimizer, which can get the updated cost. For query vector, only adding/removing tables will affect the query vector. To this end, we can leave several positions for capturing future updates of adding/deleting tables.

4. DRL FOR KNOB TUNING

Since there are hundreds of knobs in a database and many of them are in continuous space [5], the database tuning problem is NP hard and it is rather expensive to find high-quality configurations [34]. We utilize the deep reinforcement learning model, which combines reinforcement learning and neural networks to automatically learn the knob values from limited samples. Note that existing DRL models [16, 19, 12] cannot utilize the query features as they ignore the effects to the environment state from the query, and we propose a Double-State Deep Deterministic Policy Gradient (DS-DDPG) model to enable query-aware tuning.

4.1 DS-DDPG Model

The DS-DDPG model contains five components as shown in Figure 4. Table 1 shows the mapping from the DS-DDPG

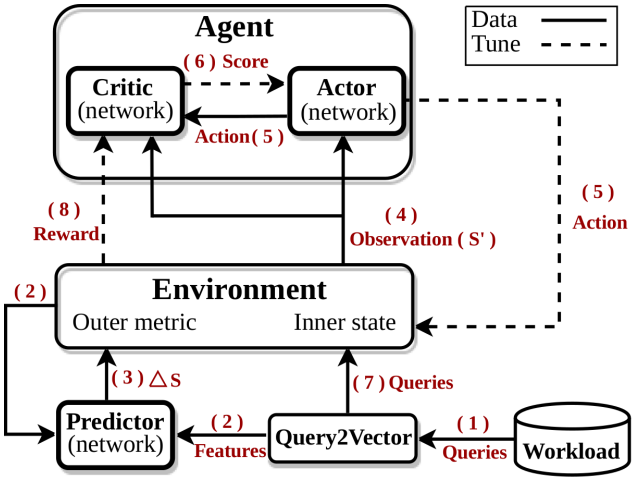


Figure 4: The DS-DDPG Model

Table 1: Mapping from DS-DDPG to Tuning

DS-DDPG	The tuning problem
Environment	Database being tuned
Inner state	Database knobs (e.g., work_mem)
Outer metrics	State statistics (e.g., updated tuples)
Action	Tuning database knobs
Reward	Database performance changes
Agent	The Actor-Critic networks
Predictor	A neural network for predicting metrics
Actor	A neural network for making actions
Critic	A neural network for evaluating Actor

model to the tuning problem. `Environment` contains the database information, which includes the inner state and the outer metrics. The inner state records the database configuration (i.e., knob configurations) which can be tuned, and the outer metrics record the state statistics (e.g., database key performance indicators), which reflect database status and cannot be tuned. For example, in PostgreSQL the inner state includes working memory, effective cache size, etc, and the outer metrics include the number of committed transactions, the number of deadlocks, etc. `Query2Vector` generates the feature vector for a given query (or a workload). `Predictor` is a deep neural network, which predicts the changes in outer metrics of before/after processing the queries. We predict ΔS because most of the outer metrics are accumulative variables (others are related to the system performance, such as the time to read a block) and their difference in values can reflect the workload's effect to the database state. Besides, predicting ΔS is much easier than S', as S' is not only related to the workload features, but current database state. `Environment` combines these changes ΔS with its original metrics S and generates the observation $S' = S + \Delta S$ to simulate the outer metrics after executing the queries. `Agent` is used to tune the inner state based on the observation S'. `Agent` contains two modules, `Actor` and `Critic`, which are two independent neural networks. `Actor` takes S' as input, and outputs an action (a vector of tuned knob configurations). `Environment` executes the query workload and computes a reward based on the performance. `Critic` takes the observation S' and the action as input, and outputs a score (Q-value), which reflects whether the action tuning is effective. `Critic` updates the weights of its neural network based on the reward value. `Actor` updates the weights of its neural network based on the Q-value. So

`Actor` generates a tuning action and `Environment` deploys the tuning action and generates a reward value based on the performance change on the new configuration. If the performance change is positive, it will return a positive reward; negative otherwise. `Critic` updates the network based on the reward. The five components work together and can automatically recommend good configurations.

DS-DDPG is an effective strategy to solve optimal problems with continuous action space by concurrently learning the Q-value function and the action policy. When the number of actions is finite, we can compute each action's Q-value and choose the action with maximal Q-value. But in continuous space, this method such as Q-learning does not work, because it's impossible to exhaustively search the space. Instead, in DDPG, we train two neural networks to adapt to continuous action space: the `Critic` network can give the Q-value for each $\langle observation, action \rangle$ and the `Actor` network updates its action policy based on the Q-value and chooses proper action according to the observation. Since neural network can perform well in high-dimensional data mapping with proper architecture design and training, DS-DDPG can also handle problems with high-dimensional input/output data. In the database tuning problem, we need to tune many knobs which are in continuous space, and thus DS-DDPG is suitable for this problem.

4.2 Training DS-DDPG

We discuss how to train the DS-DDPG model (Predictor, Actor and Critic), and Algorithm 1 shows the pseudo code.

4.2.1 Training the Predictor

Training Data T_P. `Predictor` aims to predict the database metrics change if processing a query in the database. The training data is a set of tuples $T_P = \{\langle v, S, I, \Delta S \rangle\}$, where v is a vector of a query, S is the outer metrics, I is the inner state and ΔS is the outer metrics change by processing v in a database. For each $\langle v, S, I \rangle$, we train `Predictor` to output a value that is close to ΔS.

The training data can be easily obtained as follows. Given a query workload, for each query q, we first use `Query2Vector` to generate v and obtain metrics S and state I from the `Environment`. Then we run q in the database and record the metrics change ΔS.

Training. `Predictor` is a multilayer perceptron model, which is composed of four fully connected layers: the input layer accepts the feature vector and outputs the mapped tensor (higher dimensions) to the hidden layers. These two hidden layers have a series of non-linear data transformations. The output restricts the tensor to the scale of the database state and generates a vector representing the predicted database metrics changes. The network actually represents a chain of function compositions which transform the input to the output space (a pattern) [6]. To avoid our network model from just learning in linear transformations, we add *ReLU* (a type of activation function most commonly used in neural network [1]) to the hidden layers to capture more complicated patterns. The weights in the network are initialized by the standard normal distribution.

Given a training dataset $\{(v_1, S_1, I_1, \Delta S_1), \ldots\}$, the training target is to minimize the error function, defined as

$$E = \frac{1}{2} \sum_{i=1}^{|U|} ||G_i - \Delta S_i||^2 \ . \quad (1)$$

Algorithm 1: Training DS-DDPG

Input: U: the query set $\{q_1, q_2, \cdots, q_{|U|}\}$
Output: π_P, π_A, π_C

```
Generate training data T_P;
TrainPredictor(π_P, T_P);
Generate training data T_A;
TrainAgent(π_A, π_C, T_A);
```

Function TrainPredictor(π_P, T_P)

Input: π_P: The weights of a neural network; T_P: The training set

```
Initiate the weights in π_P;
while !converged do
    for each (v, S, I, ΔS) ∈ T_P do
        Generate the output G of ⟨v, S, I⟩;
        Accumulate the backward propagation error:
          E = E + 1/2 ||G − ΔS||^2;
    Compute gradient ∇_{θ_s}(E), update weights in π_P;
```

Function TrainAgent(π_A, π_C, T_A)

Input: π_A: The actor's policy; π_C: The critic's policy; T_A: training data

```
Initialize the actor π_A and the critic π_C;
while !converged do
    Get a training data
      T_A^1 = (S'_1, A_1, R_1), (S'_2, A_2, R_2), ..., (S'_t, A_t, R_t);
    for i = t − 1 to 1 do
        Update the weights in π_A with the
          action-value Q(S'_i, A_i|π_C);
        Estimate an action-value
        Y_i = R_i + τQ(S'_{i+1}, π_A(S'_{i+1}|θ^{π_A})|π_C);
        Update the weights in π_C by minimizing the
          loss value L = (Q(S'_i, A_t|π_C) − Y_i)^2;
```

where G_i is the output value by `Predictor` for query q_i, and U is the query set.

We adopt Adam [10] to train `Predictor`. Adam is a stochastic optimization algorithm. It iteratively updates the network weights by the first and second moments of the gradients, which are computed using stochastic objective function. The training procedure terminates if the model is converged or runs a given number of steps.

4.2.2 Training the Actor-Critic Module

Training Data T_A. The agent (i.e., the Actor-Critic module) aims to judiciously tune the database configurations. Given a query workload, we randomly select a subset of queries and generate a sample workload. For the sample query workload, we generate its feature vector via `Query2Vector`, predict a database metrics S'_1 via `Predictor`, get an action A_1 via `Actor`, deploy the actions in the databases, run the database to get a reward R_1 (the reward function will be discussed later). In the next step, we get a new database metrics S'_2 by updating S'_1 using the new metrics, and repeat the above steps to get A_2 and R_2. Iteratively, we get a set of triples $\langle T_A^1 = (S'_1, A_1, R_1), (S'_2, A_2, R_2), \ldots, (S'_t, A_t, R_t) \rangle$ until the average reward value is good enough (e.g., the average reward of ten runs is larger than 10.)

Training Actor and Critic. The training of the Actor-Critic module is to update the weights in their neural net-

works. We first initiate the DS-DDPG model, including the environment, the actor policy π_A and the critic policy π_C. Then we use the experience replay method to train the actor and the critic in the reinforcement learning process.

Given a training dataset $\langle T_A^1 = (S'_1, A_1, R_1), (S'_2, A_2, R_2), \ldots, (S'_t, A_t, R_t)\rangle$, we consider (S'_i, A_i, R_i) and update `Actor` and `Critic` as follows.

(1) We update the actor policy π_A using the gradient value

$$\nabla_{\theta^{\pi_A}} \pi_A = \nabla_{A_i} Q(S'_i, A_i|\pi_C) \cdot \nabla_{\theta^{\pi_A}} \pi_A(S'_i|\theta^{\pi_A})$$

where θ is the parameters in π_A and $Q(S'_i, A_i|\pi_C)$ is the Q-value computed by `Critic`.

(2) We estimate the real action-value Y_i. We use the Bellman function [32] to compute Y_i based on the reward and Q-value, i.e.,

$$Y_i = R_i + \tau \cdot Q(S'_{i+1}, \pi_A(S'_{i+1}|\theta^{\pi_A})|\pi_C)$$

where τ is a tuning factor to tradeoff the Q-value and the reward value.

(3) We calculate the loss value L with Q and Y. `Critic` updates the weights in π_C by minimizing the loss value

$$L = (Q(S'_i, A_i|\pi_C) - Y_i)^2$$

We run the three steps for $i = t-1$ to $i = 1$.

The algorithm terminates if the model is converged (e.g., the performance improvement is smaller than a threshold); otherwise we select next training data $T_A^2, T_A^3, \cdots$.

Target network. To improve the stability of training, we can introduce two extra target actor and critic networks (whose policies are π'_A and π'_C respectively). These two networks are updated at every step and their weights (parameters of the policy) are updated slower than the normal networks. Then the weights in the normal critic network are updated by minimizing loss compared with the target:

$$L(\pi_C) = (Q(S'_i, A_i|\pi_C) - Y_i)^2$$

$$Y_i = R_i + \tau \cdot Q(S'_{i+1}, \pi'_A(S'_{i+1}|\theta^{\pi'_A})|\pi'_C)$$

Reward Function. Our reward function is designed to capture two abilities. 1) It can provide valuable feedback of the database performance; 2) It takes multiple metrics into consideration, and each metric can have different importance by assigning different weights.

Step 1. For each metric m, e.g., latency and throughput, calculate the performance change compared with that at initial time ($\Delta_{0,t}$) and that at last time ($\Delta_{t-1,t}$).

$$\Delta_{0,t} = \begin{cases} \frac{m_t - m_0}{m_0}, & \textit{the higher the better} \\ \frac{m_0 - m_t}{m_0}, & \textit{the lower the better} \end{cases}$$

$$\Delta_{t-1,t} = \begin{cases} \frac{m_t - m_{t-1}}{m_{t-1}}, & \textit{the higher the better} \\ \frac{m_{t-1} - m_t}{m_{t-1}}, & \textit{the lower the better} \end{cases}$$

Step 2. The reward function of metric m is designed as:

$$r_m = \begin{cases} ((1+\Delta_{t-1,t})^2 - 1)|1 + \Delta_{0,t}|, & \Delta_{0,t} > 0 \\ -(((1-\Delta_{t-1,t})^2 - 1)|1 - \Delta_{0,t}|), & \Delta_{0,t} \leq 0 \end{cases}$$

Step 3. The reward function R on multiple metrics is

$$R = \sum w_m r_m$$

where w_m is the weight manually assigned for metric m.

Algorithm 2: Tuning with DS-DDPG

Input: Q: a query set $\{q_1, q_2, \cdots, q_{|Q|}\}$
Output: Action A

```
V = Query2Vector(Q);
ΔS =Predictor(V);
S = Enviorment();
A =Actor(S' = S + ΔS);
Deploy A;
Run Q;
```

The reward function is similar to that in CDBTune [36].

Remark. We have three neural networks in DS-DDPG, `Predictor`, `Actor` and `Critic`. Although there is no "standard" concepts in constructing a neural network [6], we design each network by considering two factors. First, the scale of input/output vectors. Generally, the input/output sizes determine the number of neurons in each layer. For example, in PostgreSQL, `Actor`'s input size is 19 and output size is 64. So the neuron number in each layer ranges from min(input size, output size) to two times bigger than max(input size, output size). In this way, there is a procedure from expending the output space to converging to target space when the network transforms input data. Second, the uncertainty between input and output. Usually, an input is not only mapped to an output or an optimal output area, especially when the output is of high dimension. Instead, in the case where one input with multiple output values occurs in the training set, it can confuse a simple neural network and requires more layers to figure out the relations. For example, for the same observation, diverse actions outputted by `Actor` can gain similar high Q-values, which estimate the benefit of an action. To help `Actor` further discover optimal areas, we add a dropout layer after each dense layer, which randomly deactivates neurons in the upper layer to explore wider action space and cut down the possibility of over-fitting. Based on the two factors, we construct a basic architecture. The network's configuration needs to be manually tuned iteratively during training by pruning redundant nodes, expanding the network or editing the weights. We increase the value if network is slow to converge; otherwise we decrease the value. When all the networks are well trained, the DS-DDPG model is captable to adapt to new workloads (with the Predictor), database state (with the observation) and even the hardware environment (with the reward).

4.3 Tuning with DS-DDPG

Algorithm 2 shows the pseudo code of tuning with DS-DDPG. Given a tuning request (a query or a query workload), `Query2Vector` generates a feature vector. `Predictor` utilizes the feature vector and generates the predicted state change ΔS. `Environment` takes ΔS as input and generates the observation $S' = \Delta S + S$ based on its current metrics S. `Actor` takes the metrics S' as input, and outputs an action (a vector of suggested knob values). `Environment` deploys the new configurations and executes the query. For cluster-level tuning, given a tuning request (a query workload), `Query2Vector` generates a feature vector for each vector. `Vector2Pattern` predicts a pattern for each vector using the DL model. `Pattern2Cluster` clusters the queries into several groups. Next we use the above algorithm to tune each query group.

5. QUERY CLUSTERING

For workloads including both transactional queries and analytical queries [25], if the user aims to optimize the latency, we recommend *query-level* tuning; if the user aims to improve the throughput, we recommend *workload-level* tuning. For analytical only queries, we recommend cluster-level tuning to balance the throughput and latency. The key problem in the cluster-level tuning is (1) how to efficiently find the appropriate *configuration pattern* for each query and (2) how to cluster the queries based on the configuration pattern. This section studies these two problems.

5.1 Configuration Pattern

The configuration pattern of a query should include all the knobs used in the DS-DDPG model. Thus a nature idea is to use DS-DDPG to generate a continuous knob configuration and take the knob configuration as the pattern. However it is rather expensive to get the continuous knob values, especially for a large number of queries. More importantly, when we cluster the queries, we do not need to use the accurate configuration pattern; instead approximate patterns are good enough to cluster the queries.

To this end, we discretize the continuous values into discretized values. For example, we can discretize each knob into {-1,0,+1}. Specifically, for each knob, if the tuned knob value is around the default value, we set it as 0; 1 if the estimated value is much larger than the default value; -1 if the estimated value is much smaller than the default value.

To avoid the curse of dimensionality when clustering queries, we only choose knobs most frequently tuned by DS-DDPG as the features, about 20 in PostgreSQL. But the new knob space is still very large. For example, if 20 knobs are used and each has 3 possible values, then there are 3^{20} possible cases. The traditional machine learning methods or regression models are hard to solve this problem, because they either assume the labels are independent such as Binary Relevance [18] or cannot support so many labels such as Classifier chain [26].

Learning Discrete Configuration Patterns Using Deep Learning. We choose the deep learning method to map queries to discrete configuration patterns. As Figure 5 shows, the `Vector2Pattern` uses a neural network, which adopts a five-layer architecture.

The input layer takes the feature vector as input and maps it to the target knob space, in order to make the input and output in the same scale.

The second layer is designed to explore the configuration patterns. It is a dense layer with ReLU as the activation function ($y = max(x, 0)$, where x is an input feature and y is the corresponding output features, using to learn two aspects of knowledge: 1) Interaction effects; 2) Non-linear effects. Interaction effects capture the correlations among the input features. For example, feature v_i captures the value difference when other features' value changes. While Non-linear effects learns non-linear mapping relations between input vector and output vector.

The third layer is a BatchNormal layer. It normalizes the input vector in favor of gaining discretized results.

The fourth layer has the same function as the second and the value of each feature in this layer's output vector ranges from 0 to 1. But different from `Predictor` in Section 4, the last layer uses a *sigmoid* activation function $S(z_i) = \frac{1}{1+e^{-z_i}}$, where z_i is the i_{th} feature of the input vector. It takes a real value as input and outputs a value in 0 to 1. This aims to do a non-linear data transformation and at the same time keeps the features in the limited range.

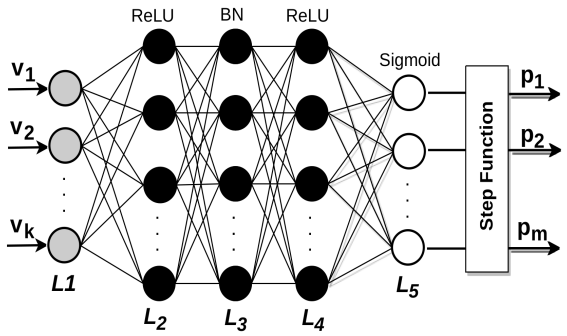


Figure 5: Architecture of the DL model

For the DL model, we also append a step function to the network's end and use the output layer as a probability distribution function: for each feature y in the output vector, the resulting bit is -1 if y is below 0.5; 0 if y equals to 0.5; and 1 otherwise. In this way, the DL model can automatically finish data transformation and discretization work.

Workflow of `Vector2Pattern`. The DL model works in 4 steps: 1) For each training sample $\langle q, p_r \rangle$, where q is a query and p_r is the real pattern that matches q, compute the feature vector v of query q; 2) Propagate these features through its network; 3) Output an estimated pattern p_e; 4) Based on the output pattern p_e and the actual pattern p_r, update the weights in the network by minimizing $|p_e - p_r|$.

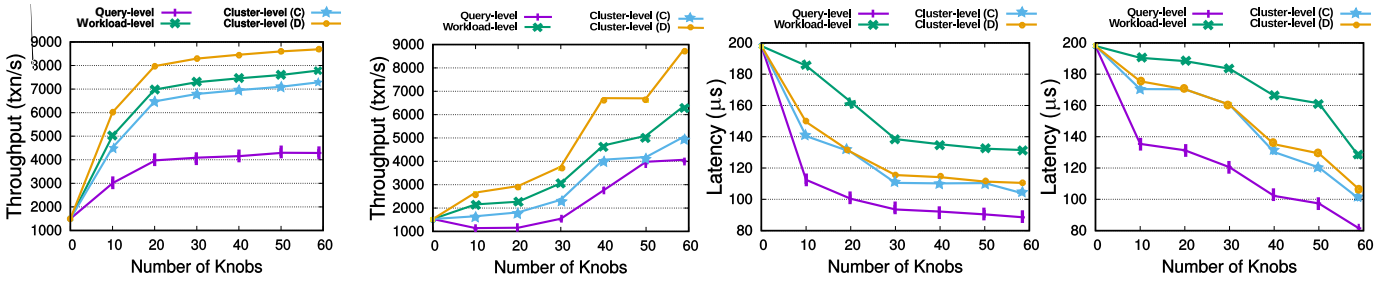
Training Step. We need to generate a large volume of samples to train `Vector2Pattern` until the performance on a new testing set is good enough (i.e., high generalization ability). Each training sample is in the form of $\langle q, p \rangle$, where q is a query statement and p is a configuration pattern under which the database can efficiently execute q. To collect these samples, we follow 3 steps: 1) Train the DS-DDPG model until it converges; 2) Select 10,000 real queries from the training data; 3) For each query q in the selected queries, use `Query2Vector` to featurize q and get v. We input the vector v into the DS-DDPG model, and get a recommended configuration to measure the performance of this query. If the performance is good enough, we discretize this configuration into pattern and the iteration terminates.

5.2 Query Clustering

After gaining the suitable configuration pattern for each query, we classify the queries into different clusters based on the similarity of these patterns. Any clustering algorithms can be used to cluster the configurations, and we take `DBSCAN` [8] as an example. Based on the configuration pattern, `DBSCAN` groups the patterns together that are close to each other according to a distance measurement and the minimum number of points to be clustered together.

6. EXPERIMENT

Out tuning system `QTune` has been deployed into Huawei Gauss database. We compare `QTune` with state-of-the-art methods [2, 36, 38]. We first evaluate our techniques, including evaluating the three types of tuning methods and the featurization methods. We then compare `QTune` with existing methods OtterTune [2], CDBTune [36], BestConfig [38]. Finally, we evaluate the generalization ability by varying different workloads, databases and hardware.



(a) IF-Throughput (b) RC-Throughput (c) IF-Latency (d) RC-Latency

Figure 6: Performance by increasing knobs in Important First (IF) and Randomly Choosing (RC) respectively when running Sysbench (RO) on PostgreSQL.

Table 2: Database information

Database	Knobs without restart	State Metrics
PostgreSQL	64	19
MySQL	260	63
MongoDB	70	515

Table 3: Workloads. RO, RW and WO denote read-only, read-write and write-only respectively.

Name	Mode	Table	Cardinality	Size(G)	Query
JOB	RO	21	74,190,187	13.1	113
TPC-H	RO	8	158,157,939	50.0	22
Sysbench	RO, RW	3	4,000,000	11.5	474,000

Database Systems. We implement the neural networks using Keras[1] with TensorFlow[2] as the backend, and use Python tools such as psycopg2, scikit-learn and numpy[3] to interact with databases and pre-process data. Since the database metrics and knobs in different database systems are different, we utilize three database systems and their related information is shown in Table 2. As restarting database is not acceptable in many real business applications, here we only use the knobs that do not need to restart databases. Note that MongoDB[4] is a document-oriented NoSQL Database. It uses json format queries rather than SQL. To run a SQL benchmark, we convert the data sets into json documents before injecting them into the database and transforms the SQL queries to json format queries.

Workload. We use three query workloads JOB[5], TPC-H[6] and Sysbench[7]. Table 3 shows the details.

Training Data. Table 4 shows the training data to train the DL model and DRL model.

Metrics. We use latency and throughput to evaluate the performance. We also evaluate the training and tuning time.

The experiments are conducted on a machine with 128GB RAM, 5TB disk, and 4.00G CPU.

6.1 Evaluation on Our Techniques

6.1.1 *Evaluation on Tuning Methods*

We compare four tuning methods, query-level, workload-level, cluster-level using DRL for tuning continuous knobs (denoted by cluster-level(C)), and cluster-level using DL model for tuning discrete knobs (denoted by cluster-level(D)). We vary the number of knobs. Here we use two methods to sort the knobs: (1) Random. We permute the knobs in a random way. If we tune k knobs, we select the first k knobs. (2) Important first. We sort the knobs based on their importance (e.g., which knobs were tuned more in the query workload). If we tune k knobs, we select the first k knobs. We conduct this experiment using JOB(RO) on PostgreSQL. Figures 6(a) to 6(d) show the results, where the point with x-axis of 0 represents the default configuration without tuning.

Table 4: The number of training samples for the DL model in query clustering, the Predictor and the Actor-Critic module in DS-DDPG.

Name	Sysbench	JOB	TCP-H
`DL`	3792	8000	40,000
`Predictor`	3792	8000	40,000
`Actor-Critic`	1500	480	300

We make the following observations from the results. First, the more knobs we use to tune, the better performance (higher throughput and lower latency) we can achieve. This is because, we have higher opportunities to use more knobs to improve the database performance. But at the same time, all methods take more training time, because they increase the network size and require to tune more parameters.

Second, cluster-level tuning achieves higher throughput than query-level and workload-level tuning. The reasons are two fold. First, cluster-level tuning can execute the query in parallel but query-level tuning cannot. Second, cluster-level tuning can provide better knob values for the queries while workload-level tuning can only provide the same knob values for all queries (which may not be optimal for most of queries). Cluster-level(D) tuning achieves higher throughput than Cluster-level(C) tuning, as continuous tuning takes more time to generate the patterns than discrete tuning.

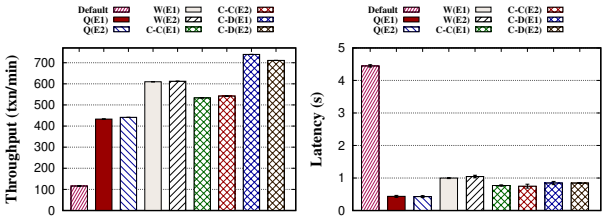
Third, query-level tuning achieves lower latency than cluster-level tuning, which in turn achieves lower latency than workload-level tuning. This is because query-level tuning can get the best knob values for each query, cluster-level tuning gets good knob values for a cluster of queries, and workload-level tuning generates the same knob values for all queries. Cluster-level (D) tuning and Cluster-level (C) tuning achieve similar latency, because the former achieves shorter tuning time but worse tuning knob values; while the latter has longer tuning time and better knob values.

Fourth, all the methods have the same performance trends on the two knob selection strategies. The importance first method has much higher performance gain than the random method, because the former first tunes the most important knobs. So using the important knobs can help the learning of the neural networks very efficiently. By comparison, randomly choosing knobs may use many knobs that have little effect on the performance or is of complex relationships with

[1]https://keras.io
[2]https://tensorflow.google.cn/
[3]http://initd.org/psycopg,scikit-learn.org,numpy.org
[4]https://www.mongodb.com/
[5]https://github.com/gregrahn/join-order-benchmark
[6]http://www.tpc.org/tpch/
[7]https://github.com/akopytov/sysbench

Database	Featurization	Tuner	Vector2Pattern	Clustering	Recommendation	Execution	Overhead
MySQL	9.37 ms	2.23 ms	0.29 ms	1.64 ms	4.36 ms	0.45 s - 262.9 s	3.8 % - 0.0068 %
PostgreSQL	9.46 ms	2.38 ms	0.39 ms	2.51 ms	5.01 ms	0.46 s - 263.3 s	4.1 % - 0.0075 %
MongoDB	13.48 ms	2.16 ms	0.36 ms	2.32 ms	4.31 ms	0.63 s - 264.5 s	3.5 % - 0.0085 %

Table 5: Time distribution of queries in JOB (RO) benchmark on MySQL, PostgreSQL and MongoDB respectively. Execution is the range of time the database executes a query. Overhead is the percentage of tuning in the total time for a query.



(a) Throughput (b) Latency

Figure 7: Performance comparison of 2 Featurization methods (E1, E2) when running JOB (RO) on PostgreSQL. (Query-level(Q), Workload-level(W), Cluster-level-C (C-C)), Cluster-level-D(C-D)

the other knobs and the database performance. It leads to longer learning time and less obvious performance gains.

6.1.2 Evaluation on featurization Methods

We evaluate two featurization methods. The first (E1) is our method in Section 3 that uses query type, tables, and costs. The second (E2) uses query type, tables, attributes, operations, and costs. Figure 7 shows the results. We find that under any tuning method, E1 and E2 achieve similar performance in throughput and latency. So we can find that adding attribute information into the feature vector does not make much difference in performance optimization. This is because the detailed attribute information is less important when tuning for all the queries together. Besides, the number of attributes in a database varies by creating or deleting tables. So to add attribute information, we need to either leave several reserved bits for attributes or re-tune the neural networks every time the attributes change, which significantly weakens the adaptivity of `QTune`.

6.1.3 Evaluation on Tuning Time

In order to better understand the distribution of execution time in a tuning and training step, we compare time consumption in each main components of `QTune`.

(1) **Featurization**: It generates the query plan using database optimizer, extracts query information and cost features from the plan and produces the feature vector.

(2) **Tuner**: The DS-DDPG model predicts the state changes, recommends a suitable configuration by the Actor-Critic module and re-configures the database.

(3) **Vector2Pattern**: The DL model transforms the input features via each layer and produces a discretized configuration pattern.

(4) **Clustering**: It divides the queries into some clusters. We use the clustering algorithm `DBSCAN`, which takes the query patterns as input and outputs some clusters.

(5) **Recommendation**: It runs the DS-DDPG model and configures the database.

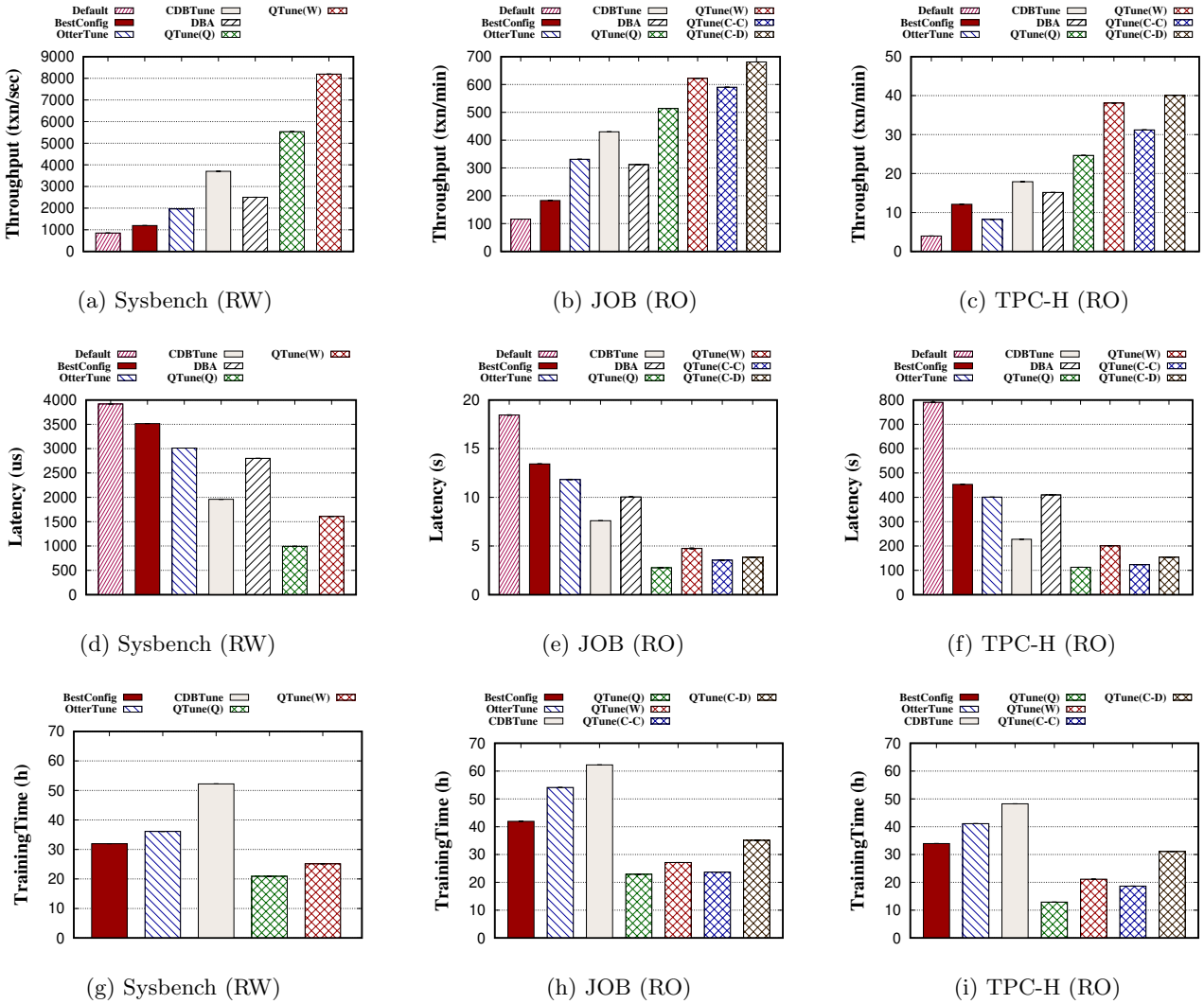
(6) **Query Execution**: It executes queries in the database.

From the results in Table 5 we can see that 1) the overhead (tuning time) takes a very small percent compared with query execution; 2) It takes `QTune` more time to tune MongoDB, a NoSQL database, compared with the other two RDBMSs, because rather than using statistics methods to estimate the execution cost, MongoDB adopts an empirical method: it actually runs different query plans in round-robin fashion and chooses the one with the best performance. So it is relatively costly to explain the queries in MongoDB.

6.2 Comparison with Existing Techniques

We compare the performance of `QTune` with 1) database default settings, 2) BestConfig [38], 3) OtterTune [2], 4) DBA, 5) the traditional RL model CDBTune [36]. BestConfig is a search-based tuning method. OtterTune is a tuning system using traditional machine learning model. For PostgreSQL, we have invited a DBA with 8 years of working experience at Huawei; for MySQL, we invited a DBA with 5 years working experience; for Mon- goDB, we have invited DBA with 2 years working experi- ence. For a new tuning requirement, DBAs took 5 days to tune. CDBTune uses a deep reinforcement learning based method to tune the database. It requires to do stress testing using query workload and takes dozens of minutes for online tuning. For each method, we conduct experiments on PostgreSQL with Sysbench (RW), JOB (RO) and TPC-H (RO) respectively. The hardware environment is Instance B in Table 6. Figures 8(a)-8(i) show the results.

We make the following observations. First, `QTune` achieves the best performance in all cases. For example, `QTune` (C-D) gains about 151.10% throughput improvement, 60.18% latency reduction and 66.15% training time reduction over that of CDBTune. The reason is as follows. CDBTune adopts the Actor-Critic module to recommend configuration, which only takes the database state as the observation but does not contain the current queries' information. So it can only indirectly learn the knowledge of the queries via the reward function. For each training cycle, our workload is randomly made up rather than repeating the same workload monotonously. So CDBTune performs much worse than `QTune`. But since CDBTune can efficiently learn from the past experience, it outperforms the other four methods. OtterTune works by mapping the workload to a workload template and generating better knob values. OtterTune utilizes Gaussian Process (GP) to map configurations and also can learn from the history, but this regression model is still too simple compared with the neural networks and cannot explore new knowledge to refine itself. Besides, OtterTune filters twice to gain the most important metrics and knobs. This filtering procedure can cause information loss, because those filtered information might be less important than the chosen ones but still takes effect on the database performance. This pipeline architecture limits the model from learning from real data. The DBAs are experienced experts, who can map the state to a typical scenario template and tune the related configurations by experience. But it's nearly impossible for humans to master the complex correlations among hundreds of knobs. DBAs usually just try



(a) Sysbench (RW) (b) JOB (RO) (c) TPC-H (RO)

(d) Sysbench (RW) (e) JOB (RO) (f) TPC-H (RO)

(g) Sysbench (RW) (h) JOB (RO) (i) TPC-H (RO)

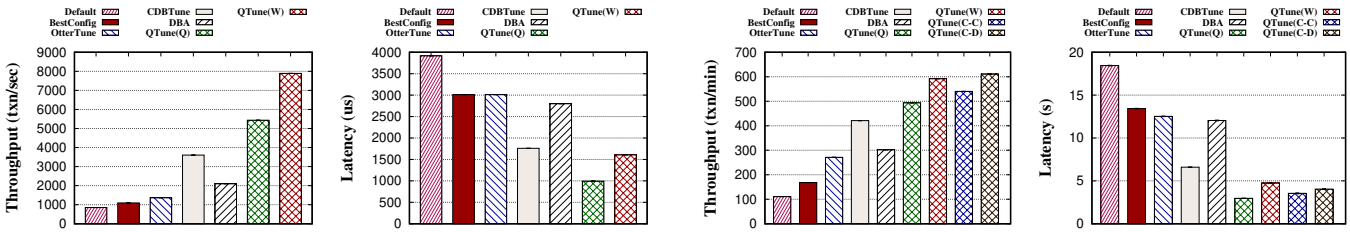
Figure 8: Comparison with existing methods Default settings, BestConfig, OtterTune, CDBTune, DBA on Sysbench (RW), JOB (RO) and TPC-H (RO) on PostgreSQL. `QTune` (Q) represents query-level tunig. `QTune` (W) represents workload-level tuning. And `QTune` (C-C) and `QTune` (C-D) indicate cluster-level tuning using Continuous Tuner and Discrete Tuner respectively.

several impactful knobs. With rich experience they can find a usable configuration in a short time, but the results are usually not good enough. Moreover, it takes DBAs very long time to figure out an ideal knob pattern. BestConfig starts by randomly choosing some knob combinations and explores from the best configuration points iteratively. Since the provided resource is limited, it usually can only gain a sub-optimal configuration. Besides, each searching period restarts and cannot utilize the past searching results. So the performance improvement is not very good sometimes.

Second, `QTune` (C-D), cluster-level tuning with discrete tuner, achieves the highest throughput, because it makes good tradeoff between providing good knobs for a group of queries and achieving high tuning time. `QTune` (Q), query-level tuning achieves the lowest latency, because it provides the best knob values for each query.

Third, CDBTune takes the longest training time. On the one hand, in each training cycle, it requires to run training examples on the databases, which is time consuming. On the other hand, it only tunes according to the database state and without filtering it utilizes all the dynamic knobs, which takes longer time to meet the performance requirements. Instead, the training time of BestConfig is controlled by the resource limits. OtterTune recommends configurations according to both the workload and performance metrics and do not need to actually run the training examples.

Fourth, our method has much larger improvement on TPC-H than JOB and Sysbench. This is because TPC-H simulates the real OLAP working scenarios and each query contains many complex operations, such as the "join". In order to support such complex workload, `QTune` has large improvement space to explore and thus gets great gains by efficiently analyzing the query characters and database states. And queries in JOB also contain many join operations and are costly to execute, and thus `QTune` still can optimize the query plan and execution procedure. But in Sysbench, the queries are simple and randomly produced based on several simple rules. They have similar structures and take few resources to finish. So the improvement is not obvious in Sysbench. Considering the other methods, we find the performance of the traditional RL model is still not bad, only worse than `QTune` on the three benchmarks. It can verify that it is feasible to use reinforcement learning in database tuning. And the performance of DBA and OtterTune is not steady.



(a) JOB(RO) to Sysb.(RW) (b) JOB(RO) to Sysb.(RW) (c) TPC-H(RO) to JOB(RO) (d) TPC-H(RO) to JOB(RO)

Figure 9: Performance when workload changes on PostgreSQL.

Because it is a bit tough for humans and the shallow machine learning methods to handle the tuning problem with different workload and data sets. For the throughput, the results are similar. And we can find from the training time that `QTune` is quite efficient to learn knowledge from a completely new starting point. But it takes relatively long time for DBA, OtterTune and the traditional RL to converge.

In summary, the DS-DDPG model suits this tuning scenario better. Rather than blindly tuning without the knowledge of the tasks to be conducted, `QTune` characterizes the queries and integrates these query features and database state into the DRL model. `QTune` can provide much better decision based on these two aspects. Besides, for workloads like OLAP, we can divide them into different clusters (cluster-level) and execute the queries by cluster.

6.3 Evaluation on Generalization

6.3.1 *Varying Different Workloads*

We verify the performance of `QTune` when workload changes. On PostgreSQL, we conduct two experiments: 1) Use the model trained on JOB (RO) benchmark to tune database on Sysbench (RW) benchmark; 2) Use the model trained on TPC-H (RO) benchmark to tune database on JOB (RO) benchmark. And we compare `QTune` with database default settings, BestConfig, OtterTune, CDBTune and DBA. Figures 9(a) to 9(d) show the results. Note that JOB queries vary in time and resource consumption, but there are only 113 queries. So we expend JOB into 10000 queries, based on the data set and existing query structures. We also extend TPC-H benchmark to generate more queries.

First, we apply these trained model for JOB (RO) to Sysbench (RW) and evaluate the performance. We make the following observations. First, `QTune` still outperforms the baselines. Query-level or workload-level tuning can adapt to query changes from Sysbench (RW). Since the optimal knob space changes after altering workload, it's tough for BestConfig to recommend suitable configurations without enough time to search. Since OtterTune relies on workload to map configuration patterns, it actually suffers most from the change. While CDBTune is free from these problems and even preforms better for JOB. This is because CDBTune does not consider queries directly and only observes the changes in database state. `QTune` performs better, because queries are vectorized by `Query2Vector` and utilized by `Predictor`. The query features include the data distribution and query costs. So our model is capable of adapting to different workloads. And even if the table schemas are different across different benchmarks, the difference in data scale and cost features can be obtained and the tuning model can recommend different configuration based on such difference.

Second, for TPC-H (RO) to JOB (RO), the results are similar. For JOB (RO), first, `QTune` performs the best among the five tuning methods. In general, TPC-H queries are much more complex than JOB queries, and `QTune` can easily utilize the knowledge learned from the query features of TPC-H to correctly estimate new feature vectors parsed from JOB, so `QTune` outperforms the others even if the workload changes. Moreover, compared with Figures 9(a)-9(b), the performance of `QTune` changes little. While the performance of BestConfig and OtterTune gets worse because it's tough for them to adapt to new workloads in a short time. The performance of CDBTune turns better because it's capable of adjusting the model according to the state change.

Table 6: Two hardware configurations

Instance	RAM (GB)	Disk (GB)	CPU (GHz)
A	16	780	2.49
B	128	5000	4.00

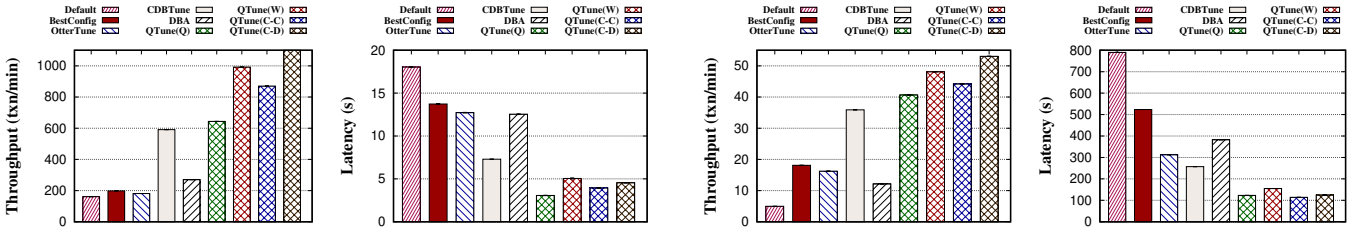
6.3.2 *Varying Different Databases*

Different databases have completely different system parameters, including different meanings, types, names and value ranges. Besides, RDBMSs may have very similar operating mechanisms, but NoSQL databases are completely different, e.g., MongoDB is based on key-value data structures and lacks of many concepts in RDBMSs such as foreign keys. To verify that `QTune` can perform well on different databases, we use three other databases to conduct experiments. We run JOB on MySQL and run TPC-H on MongoDB. For each experiment, the performance of `QTune` is compared with the other four methods. Figures 10(a)-10(d) show the results. We have the following observations.

First, `QTune` performs well on the three databases and outperforms the other four methods. This is contributed to our query-aware tuning techniques. Second, on MongoDB, `QTune` achieves the best performance improvement. This is because operating mechanisms such as index optimizations and task scheduling are not as good as RDBMS like MySQL, and PostgreSQL and MongoDB is more sensitive to different database configurations. Third, latency reduction is much less than throughput improvement, because we have large opportunity to tune throughput but less opportunity to tune latency. Fourth, `QTune` can efficiently adapt to different environments and keep in relatively good performance.

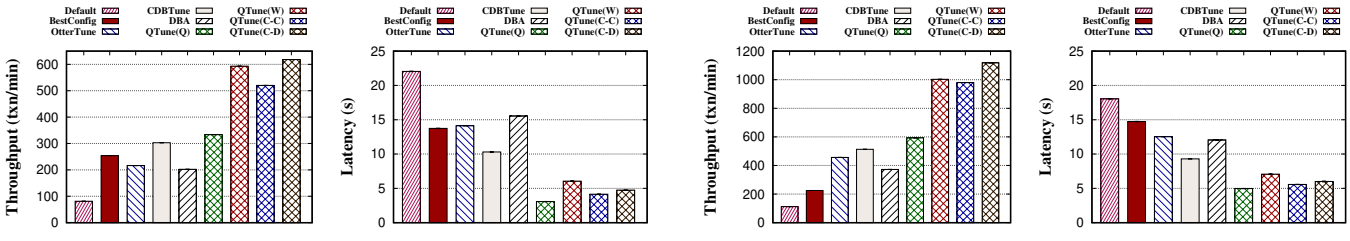
6.3.3 *Varying Different Hardware environments*

It is a challenge to adapt to new hardware environment, because the learnt knowledge of the disk size, RAM size and computing ability needs to be updated when the model is migrated to a different hardware environments. We conduct experiment to evaluate whether `QTune` can adapt to new hardware environments. We use two instances as shown in Table 6. B is better than A in each aspect. So the model trained on A has very large configuration space to explore once it's migrated to B. As shown in Figures 11(a)-11(d), M_B on A (model trained on B is used to tune on A) performs even better than M_A on A (model trained on A is used to tune on A). This is because the model trained on B



(a) MySQL (b) MySQL (c) MongoDB (d) MongoDB

Figure 10: Performance for different databases.



(a) M_A on A (b) M_A on A (c) M_B on A (d) M_B on A

Figure 11: Performance for different hardware environments.

has already explored large enough configuration space which overlaps the optimal information of instance A, or includes better configuration space than that by M_A. And for each knob value recommended by model B, if it's above the allowed range, we will change it to the boundary value. So there can be many "invalid" recommendations, but this usually won't cause resource waste. And this convergence is faster than exploring for new optimal space.

7. RELATED WORK

Database Tuning. There has been several studies on automatic database tuning, which can be divided into two categories: rule-based methods and learning-based methods.

(1) Rule-based Methods. Rule-based methods explore optimal database configurations by using rules or heuristics. iTuned [7] uses statistical methods to find most impactful knobs and tune database according to the correlations of performance and knobs. Wei et al. [33] propose a performance tuning framework which generates rules and uses these rules to conduct tuning. BestConfig [38] divides the high-dimension knob space into subspaces and iteratively chooses configurations using recursive bound and search. However they rely on either rules or history data.

(2) Learning-based Methods. Learning-based methods utilize the machine learning techniques to tune database knobs. OtterTune [2] uses a traditional ML model to map a workload to a specific configuration. However it relies on large-scale high-quality training samples. Zheng et al. [37] propose to identify key system parameters using statistical methods and utilize a neural network to match configurations for a specific workload. It also requires large volume of training samples and has the problem of overfitting. CDBTune [36] utilizes a DRL model that recommends configurations based on the database states. However, it only provides a coarse-grained tuning but cannot provide fine-grained tuning.

Reinforcement Learning. Reinforcement learning (RL) was proposed to address game theory [24, 9, 23]. RL is a framework that enables a learner (agent) to take actions in a specific scenario (environment) and learn from the interactions with the environment in discrete time steps [24]. Unlike supervised learning, RL does not need a large volume of training data. Instead, through trial and error, the agent iteratively optimizes its policy of choosing actions, with the goal of maximizing an objective function (reward). By the exploration and exploitation mechanism, RL can make a tradeoff between exploring untouched space and exploiting current knowledge. But the traditional RL approaches have difficulty in choosing the state features [22]. Mnih et al. [21] propose a DRL model combining RL with deep learning methods to solve problems with limited prior knowledge and high-dimensional state space. And many DRL algorithms, such as DQN [30] and DDPG [16], have been successfully utilized in different optimal problems [17, 3, 35, 15, 31].

There have been many researches in solving database problems with reinforcement learning [13]. Basu et al. proposed to learn the cost model with RL and use this learnt knowledge to implement index tuning [4]. Tzoumas et al. [29] assume a suitable query plan residing in the routing policies and they use a RL model to learn the policies. ReJoin [20] optimizes query plans by enumerating join orderings with RL. Sun et al. [28] proposed an end-to-end cost estimator. SageDB [11] provides a vision that the core components in a database may be replaced by learned models. Li et al. [14] propose AI-native databases that not only utilize AI to optimize databases but provide in-database AI capabilities.

8. CONCLUSION

We proposed a query-aware database tuning system `QTune` with a deep reinforcement learning (DRL) model. `QTune` featurized the SQL queries by considering rich features of the SQL queries. `QTune` fed the query vectors into the DRL model to dynamically choose suitable configurations. Our DRL model used the the actor-critic networks to find optimal configurations according to both the current and predicted database states. Our tuning system can support query-level, workload-level and cluster-level database tuning. We also proposed a query clustering method using deep learning model to enable cluster-level tuning. Experimental results showed that `QTune` achieved high performance and outperformed the state-of-the-art tuning methods.

Acknowledgement This work was supported by the 973 Program of China (2015CB358700), NSF of China (61632016, 61521002, 61661166012), Huawei, and TAL education. Guoliang Li is the corresponding author.

9. REFERENCES

[1] A. F. Agarap. Deep learning using rectified linear units (relu). *CoRR*, abs/1803.08375, 2018.

[2] D. V. Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*, pages 1009–1024, 2017.

[3] R. Ali, N. Shahin, Y. B. Zikria, B. Kim, and S. W. Kim. Deep reinforcement learning paradigm for performance optimization of channel observation-based MAC protocols in dense wlans. *IEEE Access*, 7:3500–3511, 2019.

[4] D. Basu, Q. Lin, W. Chen, H. T. Vo, Z. Yuan, P. Senellart, and S. Bressan. Regularized cost-model oblivious database tuning with reinforcement learning. *T. Large-Scale Data- and Knowledge-Centered Systems*, 28:96–132, 2016.

[5] P. Belknap, B. Dageville, K. Dias, and K. Yagoub. Self-tuning for SQL performance in oracle database 11g. In *ICDE*, pages 1694–1700, 2009.

[6] S. G. Dikaleh, D. Xiao, C. Felix, D. Mistry, and M. Andrea. Introduction to neural networks. In *CASCON*, page 299, 2017.

[7] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with ituned. *PVLDB*, 2(1):1246–1257, 2009.

[8] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.

[9] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau. An introduction to deep reinforcement learning. *CoRR*, abs/1811.12560, 2018.

[10] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[11] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. Sagedb: A learned database system. In *CIDR*, 2019.

[12] S. Krishnan. *Hierarchical Deep Reinforcement Learning For Robotics and Data Science*. PhD thesis, University of California, Berkeley, USA, 2018.

[13] G. Li. Human-in-the-loop data integration. *PVLDB*, 10(12):2006–2017, 2017.

[14] G. Li, X. Zhou, and S. Li. Xuanyuan:anai-nativedatabase. In *IEEE Data Bulletin*, 2019.

[15] K. Li and G. Li. Approximate query processing: What is new and where to go? *Data Science and Engineering*, 3(4):379–397, 2018.

[16] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[17] R. Lin, M. D. Stanley, M. M. Ghassemi, and S. Nemati. A deep deterministic policy gradient approach to medication dosing and surveillance in the ICU. In *EMBC*, pages 4927–4931, 2018.

[18] O. Luaces, J. Díez, J. Barranquero, J. J. del Coz, and A. Bahamonde. Binary relevance efficacy for multilabel classification. *Progress in AI*, 1(4):303–313, 2012.

[19] V. Maglogiannis, D. Naudts, A. Shahid, and I. Moerman. A q-learning scheme for fair coexistence between LTE and wi-fi in unlicensed spectrum. *IEEE Access*, 6:27278–27293, 2018.

[20] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In *SIGMOD workshop*, pages 3:1–3:4, 2018.

[21] V. Mnih, K. Kavukcuoglu, D. Silver, and etc. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[22] R. Munos and A. W. Moore. Variable resolution discretization in optimal control. *Machine Learning*, 49(2-3):291–323, 2002.

[23] Z. Ni, H. He, D. Zhao, and D. V. Prokhorov. Reinforcement learning control based on multi-goal representation using hierarchical heuristic dynamic programming. In *IJCNN*, pages 1–8, 2012.

[24] A. Nowé and T. Brys. A gentle introduction to reinforcement learning. In *SUM*, pages 18–32, 2016.

[25] J. S. Oh and S. H. Lee. Resource selection for autonomic database tuning. In *ICDE*, page 1218, 2005.

[26] J. Read, B. Pfahringer, G. Holmes, and E. Frank. Classifier chains for multi-label classification. *Machine Learning*, 85(3):333–359, 2011.

[27] D. G. Sullivan, M. I. Seltzer, and A. Pfeffer. Using probabilistic reasoning to automate software tuning. In *SIGMETRICS*, pages 404–405, 2004.

[28] J. Sun and G. Li. An end-to-end learning-based cost estimator. *CoRR*, abs/1906.02560, 2019.

[29] K. Tzoumas, T. Sellis, and C. S. Jensen. A reinforcement learning approach for adaptive query processing. 2008.

[30] H. van Hasselt. Double q-learning. In *NIPS*, pages 2613–2621, 2010.

[31] G. Vargas-Solar, J.-L. Zechinelli-Martini, and J.-A. Espinosa-Oviedo. Big data management: What to keep from the past to face future challenges? *Data Science and Engineering*, 2(4):328–345, 2017.

[32] C. Watkins and P. Dayan. Technical note q-learning. *Machine Learning*, 8:279–292, 1992.

[33] Z. Wei, Z. Ding, and J. Hu. Self-tuning performance of database systems based on fuzzy rules. In *FSKD*, pages 194–198, 2014.

[34] G. Weikum, A. Mönkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *VLDB*, pages 20–31, 2002.

[35] E. Wu. Crazy idea! databases reinforcement-learning research. In *CIDR*, 2019.

[36] J. Zhang, Y. Liu, K. Zhou, and G. Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *SIGMOD*, 2019.

[37] C. Zheng, Z. Ding, and J. Hu. Self-tuning performance of database systems with neural network. In *ICIC*, pages 1–12, 2014.

[38] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *SoCC*, pages 338–350, 2017.