Adaptive Multi-Model Reinforcement Learning for Online Database Tuning

Yaniv Gur
IBM Almaden Research Center
San Jose, CA
guryaniv@us.ibm.com

Dongsheng Yang
Princeton University
Princeton, NJ
dy5@princeton.edu

Frederik Stalschus
DHBW Stuttgart
Stuttgart, Germany
frederik.stalschus@ibm.com

Berthold Reinwald
IBM Almaden Research Center
San Jose, CA
reinwald@us.ibm.com

ABSTRACT

Mainstream DBMSs provide hundreds of knobs for performance tuning. Tuning those knobs requires experienced database administrators (DBA), who are often unavailable for owners of small-scale databases, a common scenario in the era of cloud computing. Therefore, algorithms that can automatically tune the database performance with minimum human guidance is of increasing importance. Developing an automatic database tuner poses a number of challenges that need to be addressed. First, out-of-the-box machine learning solutions cannot be directly applied to this problem and, therefore, need to be modified to perform well on this specific problem. Second, training samples are scarce due to the time it takes to collect each data point and the limited accessibility to query data submitted by the database users. Third, databases are complicated systems with unstable performance, which leads to noisy training data. Furthermore, in a realistic online environment, workloads can change when users run different applications at different times. Although there are several research projects for automatic database tuning, they have not fully addressed this challenge, and they are mainly designed for offline training where the workloads do not change. In this paper, we aim to tackle the challenge of online tuning in evolving workloads environment by proposing a multi-model tuning algorithm that leverages multiple Deep Deterministic Policy Gradient (DDPG) reinforcement learning models trained on varying workloads. To evaluate our approach, we have implemented a system for tuning a PostgreSQL database. The results show that we can automatically tune a PostgreSQL database and improve its performance on OLTP workloads and can adapt to changing workloads using our multi-model approach.

1 INTRODUCTION

Modern DBMSs have hundreds of configuration knobs that affect their performance. A DBMS that is not configured properly for the current workload may lead to sub-optimal performance and inefficient usage of system resources that may result in hundreds of users that are not getting the performance they need for their applications. The role of monitoring and configuring a DBMS was traditionally done by a database administrator (DBA), an expert dedicated to this task. However, nowadays, multiple DMBS instances are deployed on the cloud and each instance could host hundreds of databases, therefore, the task of monitoring and

configuring a large-scale database infrastructure requires a large number of DBAs, which would lead to high operation costs.

Over the last few years, several database vendors have identified the potential of using machine learning to automate different database tasks on the cloud, such as automatic indexing, configuration, and provisioning. A few examples include the autonomous database from Oracle [11] and the self-driving database from Alibaba [1]. The study of autonomous databases using AI is a very active research area that already yielded a large number of papers, where the most popular machine-learning paradigm in recent works is reinforcement-learning [7, 9, 14, 18]. Born as a machine-learning branch for solving complex control problems, reinforcement learning is a natural choice the automatic database tuning tasks.

One of the main challenges of operating an automatic DBMS tuning system on the cloud is the fact that the database environment is dynamic: system resources, workloads, and database size could change in the course of the day, therefore, a system for automatic tuning needs to be flexible and adapt to these changes to provide the optimal performance for a given environment state. In this paper, we address the problem of changing workloads in an online tuning setting, and we employ reinforcement learning for this task. While query-aware formulations for tuning were previously proposed [7, 16], the problem of changing workloads in an online tuning setting was not fully addressed.

Our main contributions in this paper are as follows:

- We propose a multi-model online tuning algorithm, sensitive to workload changes, that leverages multiple DDPG reinforcement learning models and selects the optimal model for evolving workloads.
- We propose a simple reward function formulation for offline and online tuning and show that it yields a more stable learning curve compared to previous art [18].
- We demonstrate the offline and online tuning algorithms on a PostgreSQL database and show that the performance of the database can be significantly improved over the baseline default performance.

2 RELATED WORK

In recent years, multiple studies have addressed the problem of automatic DBMS tuning using various machine-learning techniques. In [5] a method called adaptive sampling was used to automate the knob configuration selection by sampling from past experience and in OtterTune [16] Gaussian Process (GP) regression was used to recommend the best knob settings. Reinforcement learning over continuous action configuration space

using the DDPG algorithm was utilized in CDBTune [18]. This method was further extended in QTune [7] where a new algorithm dubbed DS-DDPG was introduced together with a database state change predictor and a query classification algorithm. The approach in this paper also uses DDPG to learn a policy function for the tuning agent, but instead of one model, uses multiple DDPG models for evolving workloads in an online tuning setting.

Reinforcement learning is not limited to database automatic tuning. In several papers, problems such as query optimization [9, 10, 12], index tuning [2, 14] and data partitioning [17] are also solved using this machine learning paradigm.

3 RL FOR DATABASE TUNING
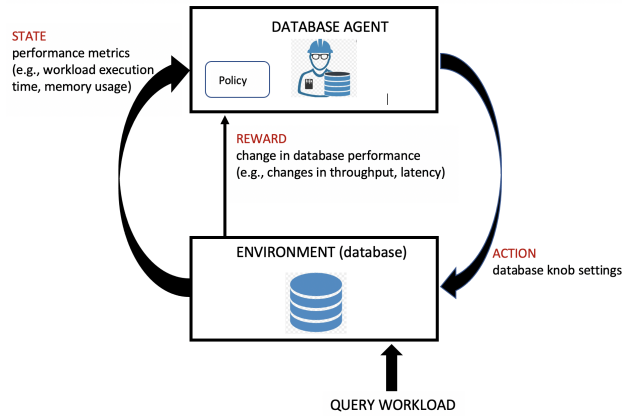
Figure 1: Reinforcement learning applied to database tuning.

A reinforcement learning system is composed of an agent and an environment. The agent takes an action on the environment and the action changes the state of the environment and generates a reward. The state and the reward are fed back into the agent that uses this input to compute the next action. The agent tries to increase the reward in every feedback loop. Reinforcement learning gained its popularity from video games, where the agent simulates a player that takes actions in the game environment and tries to win the game by maximizing a reward function.

As shown in Figure 1, in a database setting, the DBMS represents the environment and the agent represents the DBA. The state at time t, s_t, is described by a set of pre-selected database monitors (statistics collectors) and the reward, r_t, is a function of the database performance metrics (such as throughput and latency). Finally, the action a_t is the change in the configuration knobs selected for automatic tuning. The agent seeks to improve the performance of the database by changing the knob values and sensing the database state and reward.

3.1 DDPG: Knobs tuning in continuous action space

There are many algorithms for reinforcement learning, and each one of them has different variations. One criterion for choosing the right algorithm is whether the action space is discrete or continuous. In discrete action space, the agent selects one action out of a pre-defined set of actions, whereas in the latter case, the agent can pick any action from a continuous action space (e.g., a continuous value). In our case, there are potentially hundreds of knobs to be tuned, and many of them can be tuned to

any value in the allowed range of the knob. Therefore, similarly to previous works [7, 18], we selected the Deep Deterministic Policy Gradient (DDPG) RL algorithm, which was developed to tackle the challenge of control problems with a continuous action space [8]. As shown in Figure 2, the agent in the DDPG framework is composed of actor and critic neural networks. Based on the state and reward observations from the environment, the critic estimates a Q-value using the Bellman equation and target networks, and the actor computes the tuning action based on the estimated Q-value. In every tuning iteration, the actor and critic networks are updated n times using batches of samples from a replay buffer that stores the (s_t, a_t, r_t, s_{t+1}) experience tuples. The batch size and the number of update iterations are hyperparameters determined by the user of the algorithm.

Here we describe the selections we made for the key elements of the algorithm:

Reward function. To train the agent, one must define a meaningful reward function to model the database performance. Different reward function formulations had been proposed in previous works [7, 18]. These formulations take into account the initial throughput and latency and the throughput and latency in the previous iteration. In this work, we rely on these formulations, but we use a simpler function that only represents a change in the performance relative to an initial throughput and latency. Our reward function is defined as follows:

$$r_{T,L} = c_T \left(\frac{T}{T_0} - 1 \right) + c_L \left(\frac{L_0}{L} - 1 \right) \tag{1}$$

where T and L stands for throughput and latency, respectively, and $C_T + C_L = 1$. Improvement in the throughput and latency yields positive reward values, whereas negative values indicate a decline in the database performance. T_0 and L_0 are set up as baseline performance values, for example, the throughput and latency obtained by the DBMS default configuration. Due to the typical oscillatory behavior of an RL training process, omitting the dependency on previous iterations adds stability to the tuning process. As will be shown in the experiments section, this formulation together with the selection of $\gamma = 0$ (discount factor) results in a more stable tuning curve compared to [18].

Replay buffer. In a traditional replay buffer, the order in which the experience tuples are saved in the buffer has no importance, and the batch of samples is selected randomly. In this work, we use a prioritized replay buffer in which tuples are prioritized based on the agent's training loss error, therefore, samples are selected based on their importance, and the ones that are more valuable for the learning process would be selected more frequently. It was shown in [8] that prioritized replay buffers lead to a more efficient learning process.

Finally, we followed the idea proposed in [8] and added Gaussian action space exploration noise using the *Ornstein-Uhlenbeck process* [15].

3.2 Offline training

In offline training, the RL agent is trained using databases built specifically for this task and the workloads that are being submitted are configured by the user. The offline training process allows us to generate preliminary models and knob settings for specific workloads, and to tune the algorithm hyperparameters in a controlled environment.

The workload in offline training can be generated and submitted using a benchmarking tool for example, such as Sysbench
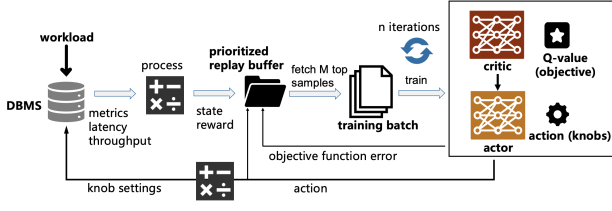
Figure 2: Database tuning with model training using the DDPG reinforcement learning algorithm.

and OLTPBench [4, 6], and the throughput and latency measures can be retrieved from the output log files generated by the tool. Once a workload is chosen, it will not be modified during the training phase. When the training phase is completed, a trained DDPG model for the workload is generated and this model can be used as a starting point for training on a different workload (transfer learning), or as a preliminary model for online tuning, a process we will describe in the next section.

The process of offline training is described in Figure 2. A submitted workload and a change in the knob settings by the action a_t cause a change in the database state and yield s_{t+1}. The throughput and latency measured over a certain time interval obtain r_t using the reward function defined in Eq. 1. The state s_{t+1} and r_t form a state transition tuple with the action a_t and the previous state s_t, and this transition tuple is added to the replay buffer. Then, a batch from the replay buffer is sampled and used to update the actor and critic networks for n iterations. Finally, a new knob settings action a_{t+1} is computed. As part of this process, the batch sample priorities are updated based on the critic loss function. The training process continues for a number of iterations that is determined by the time it takes the learning curve to stabilize.

Offline training was used in previous works for building preliminary machine learning tuning models [7, 16, 18].

3.3 Model deployment

When a trained model is deployed, the algorithm described in Figure 4 is used to recommend the knob settings. When the agent receives the database state and performance metrics, the actor predicts an action that is being translated to knob settings after a Gaussian noise is generated using the OU process and added to the action. These knob settings are then applied to the database.

4 ADAPTIVE MULTI-MODEL ALGORITHM FOR ONLINE TUNING

Tuning online is a challenging task, since workloads can change at anytime. Suppose that a database runs OLAP queries at night and OLTP queries during the daytime, its optimal memory allocation strategy would be different for each workload. Therefore, the training data from an OLAP workload cannot be used for training a model for an OLTP workload, and if the tuning system cannot detect workload changes and adapt, it would perform poorly on new workloads.

To deal with the challenge of changing workloads, we have developed a system for adaptive online tuning based on a multi-model algorithm. This algorithm tracks the database state and performance and the agent uses a set of pre-trained models and dynamically creates new models to tune the knobs if required. The performance measures (latency, throughput) in this case can

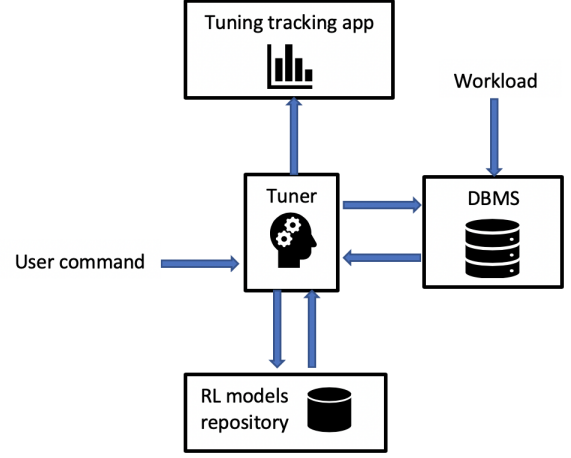be computed using database views such as *pg_stat_activity* in PostgreSQL.

Figure 3: The multi-model database online tuning system.

The main components of our system described in Figure 3 are the tuner and the RL model repository. The tuner refers to the DDPG reinforcement learning algorithm presented in Section 3. The repository of model contains a collection of RL models, pre-trained on workloads that are as similar as possible to the workloads in the deployment environment, as well as models that are created during online tuning. The models in this repository are persisted with their weights, replay buffer, and a log file that contains the knob settings that resulted in the best database performance during training. In addition, each model is accompanied by a workload representation vector that allows to retrieve the most similar models in Algorithm 1. The last component is a web-based app we implemented using Bokeh [3] that uses the log files created during training to display in real-time the database performance measures and the values of the knobs being tuned.
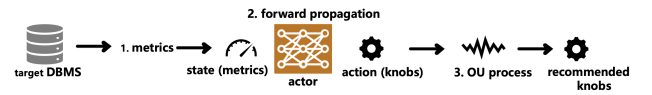

Figure 4: Knobs recommendation in deployment environment using DDPG. 1. The agent receives state and performance metrics. 2. The actor network predicts an action based on this input. 3. Exploration noise is added to the action using the Ornstein-Uhlenbeck process and a new knobs configuration is recommended.

4.1 Workload representation vectors

To generate the workload representation vectors we used an autoencoder neural network. We generated training data of state vectors that combine the database state metrics, latency, throughput, and queries per second (qps): $V = (M_0, M_1, \ldots, M_i, T, L, Q)$. These vectors were collected at different time points and for different workloads. Then, we reduced their dimension by training a simple 3-layer autoencoder and used the compressed representation from the hidden layer as the new state vectors. Since these vectors were often sparse, this procedure allowed us to obtain a

compact workload representation. The autoencoder architecture is shown in Figure 5.
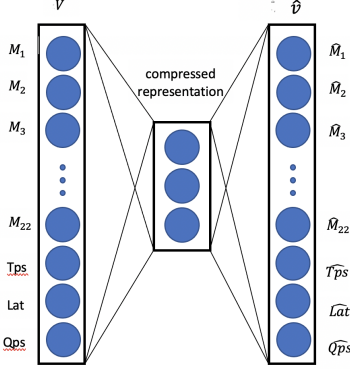
Figure 5: Workload representation dimensionality reduction using an autoencoder. V and $\hat{V}$ represents the original and reconstructed vectors, respectively. We used a 25-dimensional state vector compressed to a three-dimensional representation.

4.2 Multi-model tuning algorithm

For a given workload, Algorithm 1 selects from the N models in the repository, M_i, the ones that have high cosine similarity between the workload representation vector of the current workload, V_w and the vectors persisted with the models, V_i, and form a set of models, S. The similarity threshold is denoted by T, and was set empirically to $T = 0.8$.

Given $\tilde{N}$ models in S, we use an algorithm denoted as *Best(S)* to select the model for recommending the knob settings in every tuning iteration. The model that has the highest probability for increasing the reward (improving the database performance) is chosen to recommend the knob settings. Model selection probabilities are assigned by generating random numbers from the Beta distribution:

$$f(x; \alpha, \beta) = Cx^{\alpha-1}(1-x)^{\beta-1} \quad (2)$$

where C is a normalization constant and α, β are the distribution shape parameters. For values of $\alpha = \beta$, the distribution is symmetric and the mean is at the center of the distribution. When $\alpha > \beta$, the mean moves to the right side of the axis, and the random generator has a higher probability of yielding values larger than 0.5. We use α and β to compare the performance of the models, such that the model that performs better, has a higher α value, and therefore a higher probability to be selected in the next tuning iteration. When comparing two models, α and β counts the number of times each model produced a reward higher than the average reward from the beginning of the workload cycle to the point of measurement.

The models from the repository would always compete against a model trained from scratch (fresh model) to guarantee that the best model is being selected in cases where the model retrieved from the repository does not perform optimally. If a model M from the repository was selected and fine-tuned during the online tuning process, its fine-tuned version M' would be persisted instead of model M. The process continues as long as the online tuning phase runs. If a workload shift is detected, the model that was selected the highest number of times to predict the action within the workload cycle is persisted in the models' repository.

A model selected from the repository and updated during training will be persisted with the compressed workload representation vector, the updated weights, the replay buffer experience, and the best knob settings.

Algorithm 1 Adaptive multi-model algorithm for online tuning

---
while Tune **do**
  $S = []$
  for i=0 to N **do**
    if $\cos(V_i, V_w) > T$ **then**
      $S.append[M_i]$
    end if
  end for
  $S.append[M_{new}]$
  while True **do**
    Compute a_t using $Best(S)$ and apply it
    Wait
    Collect s_{t+1} and r_t and update the models in S
    if workload shift **then**
      Persist $Best(S)$
      Break
    end if
  end while
end while

---

5 EXPERIMENTS

In the following section, we evaluate the offline tuning algorithm and our online tuning algorithm on PostgreSQL. In the offline phase, we use the Sysbench benchmarking tool [6] to submit queries and measure the throughput and latency. Sysbench is a multi-threaded configurable benchmarking tool for OLTP workloads, where the major ones are OLTP Read/Write (R/W), OLTP Read-Only (R/O) and OLTP Write-Only (W/O). The workloads are composed of SELECT, INSERT, DELETE and UPDATE queries, where the number and mixture of the queries in each workload can be modified via command line or by modifying a Lua script that defines the workload. In addition, Sysbench allows the user to control the benchmarking duration time, the size of the database, and the number of workers.

5.1 Single-model: Static workload

In this case, a single RL model is trained and is responsible for tuning the database, regardless of the workload that is being submitted to the database. This approach works well if the database environment is relatively stable and workloads are not changing. However, if workloads are changing, the model needs to adapt to a new workload when a workload shift occurs, and it uses its past experience to recommend knobs for the new workload.

In the first experiment, we demonstrate the performance of the RL agent using the DDPG algorithm described in Figure 2. This experiment demonstrates offline training using OLTP R/W workload. We ran the algorithm for 150 episodes with the main hyperparameters setup as follows: $\gamma = 0$ (discount factor), $\sigma = 0.2$ (OUProcess noise variance), and replay buffer sampling batch size of 32 samples. To represent the DBMS state we picked 22 PostgreSQL metrics from 3 different views that provide statistics at the instance and database levels: *pg_stat_bgwriter*, *pg_stat_database*,

and *pg_stat_database_conflicts*. These views monitor various database elements such as checkpoints, buffers, deadlocks, and transactions' activity. To get the reward in the offline tuning stage, we used the throughput and latency calculated by Sysbench.

Based on experiments with different number of knobs, and expert blogs on tuning PostgreSQL (e.g., [13]), we have selected for tuning 16 knobs that had the most impact on the database performance. These knobs control various aspects of the database, such as working memory (e.g., *work_mem*, *maintenance_work_mem*), checkpoints (e.g., *checkpoint_segments*, *checkpoint_timeout*), deadlocks (*deadlock_timeout*), and auto_vaccum (*autovacuum_cost_delay*, *vacuum_cost_limit*). They do not require database restart to be updated, one of the criteria for selecting them.
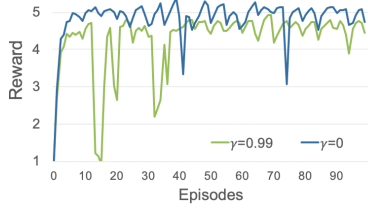
Figure 6: Reward function comparison for the OLTP R/W workload experiment. The green reward curve was obtained using the CDBTune reward [18] with $\gamma = 0.99$. The blue reward curve was obtained using our simplified reward function with $\gamma = 0$.

In all the experiments described in this section, the tuning starting point was the database default configuration. As Figure 7 shows, the default performance for OLTP R/W was a throughput of approximately 2000 TPS and a latency of 80 ms. As the agent explored different knob settings, we observed a 5x improvement in throughput and 8x improvement in latency. In the first iterations, the performance significantly oscillated, but the magnitude of the oscillation decreased as the agent learned the right policy for tuning the knobs. In the OLTP R/W experiment, we also compared the simplified formulation of the reward function (Eq. 1) to the formulation and γ parameter setup in [18] and observed that the simplified formulation with $\gamma = 0$ resulted in a more stable learning curve and a higher reward value (better database performance). The reward curves of the first 100 iterations are shown in Figure 6.

The offline tuning process can be repeated with any workload, such as OLTP W/O, and the models trained in the offline training phase can be used as initial pre-trained models in the multi-model online tuning experiment we describe next.

## 5.2 Single and multi-model approach: Changing workloads

In this experiment, we used the same knobs selected for tuning in the offline training experiment and the same 22 database metrics were used as state indicators. We created an environment in which two alternating Sysbench workloads are submitted to the database: R/W and W/O workloads and we compared the single-model, multi-model, and default database performance over 3 alternating workload cycles. At each time point in Figure 8, a workload is submitted, the state and performance metrics are collected, the RL models are updated, and finally, a knob changing action is computed by the agent to update the knobs using Algorithm 1. Each time point takes approximately 60 seconds to

(a) Throughput
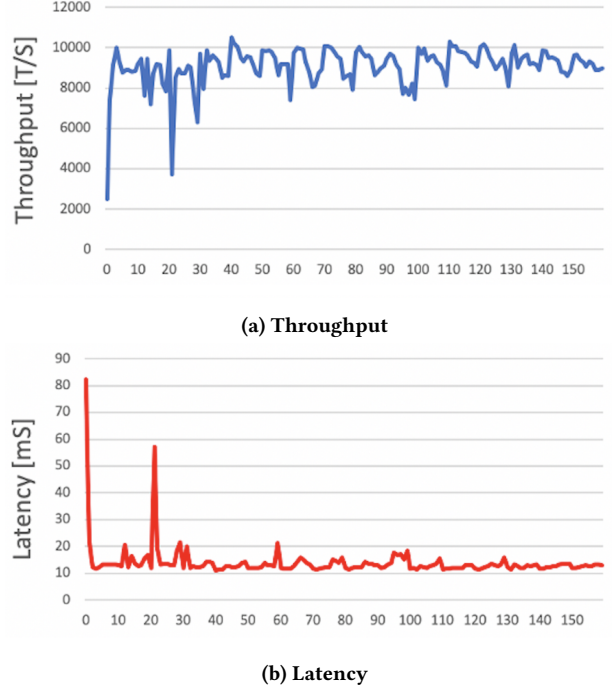
(b) Latency

Figure 7: Offline training: Throughput and latency plots of a PostgreSQL database tuned on an OLTP R/W workload.

complete, hence, workloads are changing approximately every 60 minutes. We used a three-dimensional compressed representation of the workload vectors, obtained by the autoencoder, to detect workload changes by measuring the distance between vectors at adjacent time points and identifying large deviations. This representation was also used to select the most similar models using cosine similarity. The knob settings saved with the most similar model were applied to the database and used as a starting point for tuning.

As shown in Figure 8, the multi-model algorithm performed far better than the baseline default configuration performance, increasing the throughput of the R/W workload by a factor 2x and the throughput of the W/O by a factor of 5x. The single-model approach, on the other hand, produced inferior performance compared to the multi-model approach and was less stable. In the second cycle, for instance, presumably due to bad knob settings, the throughput dropped to approximately 300 TPS, and the latency jumped to very high values. We assume that this is related to the fact that the replay experience (state, action, reward transitions) collected in the first cycle could not be properly leveraged by the agent for learning a good configuration in the next cycle. This could be related to the fact that the single model approach uses a single replay buffer for multiple workloads, therefore, experiences from different workloads with different reward scales are mixed, and similar experiences are mapped to completely different rewards. Therefore, the agent cannot learn a good policy for tuning the knobs. In the multi-model approach, each model was trained on a different workload and used experiences unique to the workload it was trained on. This models' separation helps the agent learn the right policy for a particular workload.

(a) Throughput
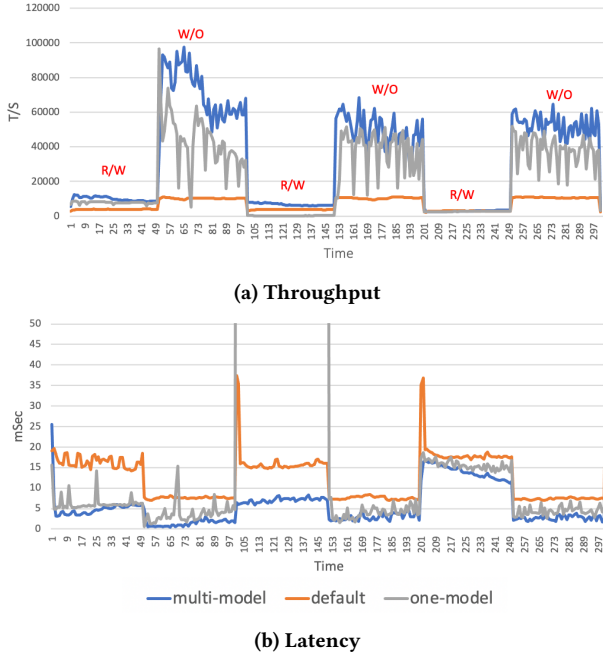
(b) Latency

Figure 8: Online tuning: Throughput and latency plots of a PostgreSQL database in changing workloads environment. The tuning results were obtained using the one- and multi-model tuning algorithms, and the default knob settings. The workloads alternate between OLTP R/W and OLTP W/O.

A potential pitfall of the multi-model algorithm is the fact that many new models can be created and lead to models' explosion and a large number of competing models for every workload. However, what we observed in our experiments, is that the pre-trained models were superior to newly created models, and therefore, new models were not persisted. Therefore, we believe that eventually, the number of models in the repository would be similar or very close to the number of different workloads the agent is tuning.

6 DISCUSSION

Automatically tuning a database on-premise or in cloud environment using reinforcement learning, or a different AI technique, poses multiple challenges. One of these challenges is the fact that the database environment is constantly changing, often in the course of the day. Changes in the database size, available system resources, and workloads may affect the performance of the database at any given time point and require an adaptive algorithm that is able to sense these changes and yield optimal performance for a given environment state. In this work, we explored the effect of changing workloads on database performance and proposed an adaptive algorithm that leverages multiple DDPG reinforcement learning models to optimize the performance for each workload. Preliminary results presented in this paper on a PostgreSQL database showed that the multi-model approach has an advantage over the single-model approach in which one model is continuously trained and needs to adapt to new workloads, similar to the approach presented in [18]. Using the multi-model approach, the algorithm was able to utilize past experiences from models trained on similar workloads and improved the database default

configuration throughput by a factor of 2x when tested on an OLTP R/W workload and a factor of 5x when tested on an OLTP W/O workload.

In future work, we will explore other workload types (e.g., OLAP), and we will address the problem of fluctuating performance due to other factors, such as resource elasticity.

REFERENCES

[1] Alibaba. 2020. Alibaba self-driving database. https://www.alibabacloud.com/product/das/
[2] Debabrota Basu, Qian Lin, Weidong Chen, Hoang Tam Vo, Zihong Yuan, Pierre Senellart, and Stéphane Bressan. 2016. *Regularized Cost-Model Oblivious Database Tuning with Reinforcement Learning.* Springer Berlin Heidelberg, Berlin, Heidelberg, 96–132. https://doi.org/10.1007/978-3-662-53455-7_5
[3] Bokeh Development Team. 2020. Bokeh: Python library for interactive visualization. https://bokeh.org/
[4] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (Dec. 2013), 277–288. https://doi.org/10.14778/2732240.2732246
[5] S. Duan, Vamsidhar Thummala, and S. Babu. 2009. Tuning Database Configuration Parameters with iTuned. *Proc. VLDB Endow.* 2 (2009), 1246–1257.
[6] Alexey Kopytov. 2020. sysbench. https://github.com/akopytov/sysbench/
[7] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2118–2130. https://doi.org/10.14778/3352063.3352129
[8] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. Continuous control with deep reinforcement learning.. In *ICLR*, Yoshua Bengio and Yann LeCun (Eds.).
[9] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management - aiDM'18* (2018). https://doi.org/10.1145/3211954.3211957
[10] Ryan Marcus and Olga Papaemmanouil. 2018. Towards a Hands-Free Query Optimizer through Deep Learning. arXiv:cs.DB/1809.10212
[11] Oracle. 2020. Oracle autonomous database. https://www.oracle.com/autonomous-database/
[12] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2018. Learning State Representations for Query Optimization with Deep Reinforcement Learning. arXiv:cs.DB/1803.08604
[13] Percona. 2018. Tuning PostgreSQL. https://www.percona.com/blog/2018/08/31/tuning-postgresql-database-parameters-to-optimize-performance/
[14] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. The Case for Automatic Database Administration using Deep Reinforcement Learning. arXiv:cs.DB/1801.05643
[15] G. E. Uhlenbeck and L. S. Ornstein. 1930. On the Theory of the Brownian Motion. *Phys. Rev.* 36 (Sep 1930), 823–841. Issue 5. https://doi.org/10.1103/PhysRev.36.823
[16] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 1009–1024. https://db.cs.cmu.edu/papers/2017/p1009-van-aken.pdf
[17] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Farooq Minhas, Umar, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning Data Layouts for Big Data Analytics. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (May 2020).
[18] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 415–432. https://doi.org/10.1145/3299869.3300085