Automatic Parameter Tuning for Big Data Pipelines with Deep Reinforcement Learning

Houssem Sagaama [§]
R&D department
EURA NOVA
Tunis, Tunisia
0000-0001-8760-3557

Nourchene Ben Slimane[§]
R&D department
EURA NOVA
Tunis, Tunisia
0000-0003-0434-310X

Maher Marwani
R&D department
EURA NOVA
Tunis, Tunisia
0000-0001-7792-7857

Sabri Skhiri
R&D department
EURA NOVA
Mont-Saint-Guibert, Belgium
0000-0002-0664-5788

***Abstract*—Tuning big data frameworks is a very important task to get the best performance for a given application. However, these frameworks are rarely used individually, they generally constitute a pipeline, each having a different role. This makes tuning big data pipelines an important yet difficult task given the size of the search space. Moreover, we have to consider the interaction between these frameworks when tuning the configuration parameters of the big data pipeline. A trade-off is then required to achieve the best end-to-end performance. Machine learning based methods have shown great success in automatic tuning systems, but they rely on a large number of high quality learning examples that are rather difficult to obtain. In this context, we propose to use a deep reinforcement learning algorithm, namely Twin Delayed Deep Deterministic Policy Gradient, TD3, to tune a fraud detection big data pipeline. We show through the conducted experiments that the TD3 agent improves the overall performance of the pipeline by up to 63% with only 200 training steps, outperforming the random search on the high-dimensional search space.**

***Index Terms*—Auto-tuning system, Big Data Pipelines, Deep Reinforcement Learning, Actor-Critic, Performance Optimization**

I. Introduction

As data volumes continue to grow at a breakneck pace, organizations are using big data pipelines to unleash the power of their data and respond faster to their demands. These big data pipelines are composed of different frameworks with different purposes such as ingestion, processing, and storage, and each of these frameworks comes with a large number of configurable parameters that need to be tuned to get the best performance (throughput and latency). For example, Apache Spark comes with over 180 configurable parameters.

Manually tuning big data pipeline parameters through trial and error is not the best option, as it is time consuming and inefficient given the huge search space and complex dependency between parameters. In addition, it requires in-depth knowledge of the underlying frameworks used in the big data pipeline. Therefore, automatic parameter tuning of big data pipelines becomes a necessity to get the most out of their performance.

Traditional tuning approaches, such as machine learning and search methods, have several limitations. On the one hand, machine learning approaches require a large number of high-quality samples to train an accurate performance prediction model, which is costly and time-consuming. Therefore, there is a need to find a trade-off between model accuracy and the cost of data collection. On the other hand, search-based models do not effectively exploit previously collected data and require a large amount of data and a large number of experiments. To address these challenges, we propose to use deep reinforcement learning to tune a big data pipeline as it is based on a trial-and-error method that uses a few samples to obtain the optimal configuration that achieves the best performance of the big data pipeline. Reinforcement learning has already been used in the task of self-tuning cloud databases or big data frameworks, however, to the best of our knowledge, optimizing the performance of a big data pipeline with a reinforcement learning agent has not yet been proposed in the literature. To this end, we propose a representation of performance metrics applied to big data pipelines and implement the twin delayed deep deterministic policy gradient, TD3, to optimize a pipeline for a fraud detection application. We show through a set of conducted experiments that TD3, regardless of the replay strategy implemented, improves the performance of the application by up to 63% in comparison with default setting, after only 200 training steps, outperforming the random search agent.

The rest of the paper is organized as follows: in section II, we first discuss existing auto-tuning methods and their limitations in the context of pipeline performance optimization and then present the performance function used to evaluate the performance of big data pipelines. Next, in Section III, we outline the state of the art of reinforcement learning algorithms and explain the choice of TD3. In Section IV, we describe the fraud detection use case used across the experiments and show the performance of the TD3 agent to tune it. Lastly, we conclude the paper in Section V and present future improvements to our work.

II. Auto-Tuning Big Data Frameworks

A. Related Works

Existing auto-tuning systems are based on different approaches that can be grouped into five categories. Rule-based methods essentially rely on domain expert guidance [1]–[3].

Supported and funded by the Walloon region, Belgium.
[§]Equal contribution

978-1-6654-2744-9/21/$31.00 ©2021 IEEE

They are simple to implement, but require in-depth knowledge of the system's internals. The disadvantage of this approach is that it takes a lot of time to follow the guidance structure. Although choosing a few parameters to tune minimizes the computational time, this can present a risk of performance degradation due to bad configurations.

Model-based methods are another category that uses cost-based analytical designs to profile, predict, and optimize the performance of specific frameworks. They are computationally practical and can provide predictions with satisfactory accuracy. But it remains difficult to capture the complexity of the system's internal and runtime components. Examples of model-based systems can be found in [4]–[7].

Simulation-based methods use a simulated environment to learn and build models to predict the performance in different environments [8]–[11]. They provide insights into the application's runtime characteristics at a reasonable cost and in a safe manner. Nevertheless, they are inefficient due to the complexity of building a simulator, which requires extensive knowledge of the system internals and the application workload.

Search-based methods as in [12]–[16], consist of reproducing a set of runs on the same workload with different configuration settings until they converge to a suitable setting. Although they are simple to run and do not require much knowledge of the system internals, they require a large amount of data and a high number of experiments. In addition, the higher the dimension of the search space, the more difficult to find the optimal configuration.

Finally, learning-based methods rely on machine learning algorithms to train a performance prediction model which is used later to find the optimal configuration using a search model such as random search and genetic algorithm [17]–[21]. With these techniques, no specific information is required about the internal elements of the system, and relatively good results can be obtained. However, some algorithms may require a large training data set. In addition, choosing the right model and appropriate hyperparameters can be a difficult task.

Fig. 2 summarizes the differences between the different tuning methods and their evolution to cover more parameters while decreasing the required knowledge about the system under tuning.

Although most of these methods can successfully be applied

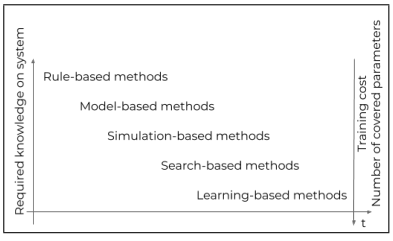

Fig. 1. Evolution of tuning methods over time

to tune a given framework and potentially lead to good results, some of them have limitations when it comes to tuning a multi-framework system. In fact, assuming that rule-based methods rely on the expertise of the domain specialists and require in-depth knowledge of the system to be tuned, it is impossible to cover the number of frameworks in the pipeline with the required advanced knowledge. Not to mention the fact that rule-based methods can only cover a small number of parameters, making them unsuitable in the context of a multi-framework parameter tuning.

This is also the case for the model-based methods. They are not able to cover the number of configurable parameters in the pipeline. Another problem that can be raised with model-based methods is their lack of sustainability. In fact, these methods are highly dependent on the version of the system under tuning, upgrading the framework to a new version will potentially invalidate the model. This will be even more complicated for a multi-framework model.

Regarding simulation-based tuning methods, the main limitation that can be pointed out is the complexity of designing a simulation environment for multiple frameworks while respecting, not only the specificity of each framework that requires a deep knowledge of the system internals, but also the relationship between the different frameworks in the pipeline.

Finally, search-based methods can be applied to tune a big data pipeline, but they require a large amount of data to converge to a good performance. This is mainly due to the highly dimensional search space of the pipeline being tuned. That being said, in this work we are interested in using learning-based methods, in particular those based on the trial-and-error learning strategy, which do not require any prior knowledge of the environment, nor a lot of training data to converge to good performance.

B. Problem Statement

Consider a function $Perf(c, r, w, d)$ which its output is a measure of the performance of a given application running on the big data pipeline. $Perf$ takes as input the configuration c, the hardware resources r, the workload w, and the input data d. It is a function of the two metrics of reference regarding big data performance, i.e. the throughput and the latency over the pipeline. We represent the end-to-end latency, L_g, as the sum of the individual latency values for every framework in the pipeline i, L_i, (1), and we represent the end-to-end throughput, T_g, as the geometric mean of the individual frameworks throughput values, T_i, (2) since the individual throughput values can be in significantly different ranges depending on the corresponding framework and the job being run.

$$L_g = \sum_i L_i \tag{1}$$

$$T_g = \sqrt[n]{\prod_{i=1}^{n} x_i} \tag{2}$$

Based on these definitions, $Perf$ will be formulated as follows:

$$Perf(c, r, w, d) = \frac{L_g}{T_g} \tag{3}$$

Finally, we can formulate the tuning problem as an optimization problem where we maximize the performance function illustrated in (3) as follows:

$$c^* = \arg\max_{c} \ Perf(c, r, w, d) \tag{4}$$

In this work, the hardware resources r and the workload w are constant. We study only the impact of tuning the configuration c with different data sizes d.

III. Reinforcement Learning

In reinforcement learning, the agent seeks to maximize a numerical reward signal while learning to map states to actions [22]. Particularly, RL online methods rely on the concept of trial-and-error with a direct interaction between the agent and the environment in which it evolves. As shown in Fig. 2, at every step t the agent observes the current state S_t, and takes an action A_t, after which it will receive a reward R_t from the environment, depending on the quality of this action at the state S_t, and observe the new state S_{t+1}. The agent performs

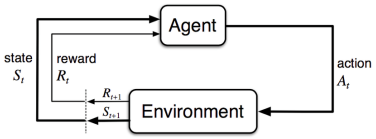

Fig. 2. The agent-environment interaction in a RL system [22].

actions according to a policy that it will try to optimize over the iterations. A policy defines the agent's behavior. It is a correspondence between the state space and the action space. The reward signal indicates immediately how good is the action that was taken by the agent. But since the goal is to maximize the long-term reward, the agent must know how to estimate the value of a given action at a given state. The value function indicates what is good in the long run. The value of a state is basically a prediction of the total amount of reward that could be accumulated in the future if the agent starts from that state. Unlike rewards that are directly given by the environment, values must be frequently re-estimated based on the sequence of observations made by the agent over the iterations. Value estimation is considered to be the most challenging part of RL algorithms, along with the exploration/exploitation trade-off. In fact, at each step, the agent can choose either to maximize the immediate reward (exploitation behavior) or to try a new action that might later lead to a better reward (exploration behavior). A good balance between both strategies is crucial for the agent to reach the best policy.

A. Problem Formulation

An optimization problem can be translated into a reinforcement learning one after an efficient representation of the RL problem components. We represent the different components of our RL tuning system as follows:

a) The State Space: in this work, the state is represented by the set of metrics that the agent will read via the monitoring tool for the different jobs running in the pipeline.

b) The Action Space: the tuning agent will recommend a configuration setting for the whole list of tunable parameters that it will receive. An action is a simultaneous association of N values to the N parameters to configure with respect to their corresponding ranges. In this work, the total number of parameters under tuning is 46 (see table I for more details)

TABLE I
Number of tuned parameters in the pipeline

Component of the pipeline	Number of parameters
Spark	20*
Cassandra	14
Kafka	12

*Including the Spark-Cassandra connector parameters.

c) The reward function: The reward is a numerical signal perceived by the agent after every recommendation. We define the reward function as in (5) with respect to the initial performance $Perf_{init}$ obtained with the default setting, and to both the current and past performances of the agent, $Perf_t$ and $Perf t-1$ respectively.

$$reward = \begin{cases} 0 & \text{if } Perf_t = Perf_{init} \\ -100 & \text{if bad config} \\ \frac{Perf_t - Perf_{t-1}}{|Perf_t - Perf_{init}|} & \text{otherwise} \end{cases} \tag{5}$$

The agent perceives a negative reward for each bad configuration. A recommended configuration is considered to be erroneous if a) the execution time of any of the jobs running in the pipeline exceeds the time limit, or b) the configuration results in an error in one of the applications (e.g. a memory error).

B. The Tuning Agent

Different RL approaches have been applied to the autotuning problem. For instance, MonkeyKing [21] applies value-based RL algorithms to autotune Apache Spark. DQN [23] was proposed to address the challenge of approximating the optimal action-value function (6) by using deep neural networks.

$$Q^*(s,a) = \max_{\pi} \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + .. \ | s_t = s, a_t = a, \pi] \tag{6}$$

DQN was the first to introduce the replay memory buffer to avoid the correlation of training data, which is likely to lead to poor convergence of the network. To solve the problem of non-stationarity due to the constant update of the targets during the training, DQN introduced the target networks that are less frequently updated to allow for more stationarity. But since the target networks are not updated frequently enough, the target values will be inaccurate during the time between two successive updates, which makes the convergence of DQN slow. Moreover, although the replay buffer helped solve the data correlation problem, it should be mentioned that with a

large buffer the agent is more likely to pull old transitions that were generated by a poor policy to train the network. The Prioritized Experience Replay, PER, has been proposed as an alternative in [24]. It considers replaying important transitions more frequently, instead of recurrently using obsolete ones.
In a different approach, CDBTune [18] and QTune [20] implemented an actor-critic method, namely Deep Deterministic Policy Gradient, DDPG, to auto-tune cloud databases. Actor-critic methods learn a value-function (like value-based methods) and a policy (like policy gradient methods) at the same time, thus combining advantages of both methods. Since DQN is more suitable for discrete action spaces, DDPG can be perceived as an extension of DQN to continuous action spaces [25]. In fact, like DQN, DDPG uses the replay memory buffer to store past experiences and learn off-policy. It also uses the target network to bring stability to the learning process. But, instead of simply updating the target network with a fixed frequency, DDPG uses the sliding average (7) for both actor and critic. Doing so will make sure the target network (8) is still "behind" the trained network, but not as much as with DQN.

$$\phi_{targ} \leftarrow \rho\phi_{targ} + (1-\rho)\phi \tag{7}$$

$$y(r, s', d) = r + \gamma(1-d)\max_{a'} Q_\phi(s', a') \tag{8}$$

As for the policy learning, DDPG trains a deterministic policy to maximize the Q-value. Since it's an off-policy algorithm, the behavior policy is separate from the target policy. It is updated by one step of gradient ascent (9), while the Q-function is updated by one step of gradient descent with a batch of transitions B (10).

$$\nabla_\theta = \frac{1}{|B|}\sum_{s\in B} Q_\phi(s, \pi_\theta(s)) \tag{9}$$

$$\nabla_\phi = \frac{1}{|B|}\sum_{(s,a,r,s',d)\in B} (Q_\phi(s,a) - y(r,s',d))^2 \tag{10}$$

For a better exploration strategy, DDPG adds noise to actions during training. It can also be kept fixed or reduced over the course of training as well. DDPG is a good alternative to DQN, but it has some limitations though. This is because, in addition to being sensitive hyperparameters, the learned Q-function is also very likely to start over-estimating the Q-values throughout the training, which makes the policy exploit the errors in the Q-function. Twin Delayed Deep Deterministic Policy Gradient, TD3 [26], solves these limitations in the same way as Double DQN [27] by adopting two Q-functions instead of one, and using the smaller value in the target to avoid the risk of over-estimation (11). TD3 learns its Q-functions in the same way that DDPG learns its single Q-function.

$$y(r, s', d) = r + \gamma(1-d)\min_{i=1,2} Q_{\phi_{i,targ}}(s', a'(s')) \tag{11}$$

TD3 also performs target policy smoothing by adding additional noise to actions during training to avoid having peaks of Q-values that are most likely to be exploited by the policy and to lead to incorrect behavior like for DDPG. Actions that will be used for the Q-learning target are based on the target policy $\pi_{\theta_{targ}}$ with a clipped noise added on each dimension (12). The resulting target action is then clipped to lie in the valid action range.

$$a'(s') = clip(\pi_{\theta_{targ}}(s') + clip(\epsilon, -c, c), a_{Low}, a_{High}), \quad \epsilon \sim N(0, \sigma) \tag{12}$$

Finally, the policy is trained following the same rule as DDPG, except that for TD3 the updates are less frequent than for DDPG. This increases the stability of the learning process.

In this work, we implement a TD3 agent to auto-tune a pipeline of Big Data frameworks. In fact, considering that value-based methods are only suitable for discrete actions and that our action space is a high-dimensional continuous space, its discretization will not only lead to a loss of information caused by the binning, but it will also drastically increase the complexity of our optimization problem (*the curse of dimensionality*). DQN and its variants are therefore not applicable for our tuning problem.
Since DDPG, the actor-critic method, showed good results in a different tuning task, and given that TD3 is supposed to improve performance over DDPG, we propose to implement TD3 for our tuning task.
In our implementation, for a better exploration at the beginning of the training, the agent takes random actions for a fixed number of steps sampled over valid actions. After that, the agent returns to normal TD3 exploration by adding noise to actions at training time. The agent interacts with the Big Data environment via the RL environment interface.
We compare the performance of the TD3 agent with its two replay strategies with random search. We express the performance improvement rate as the obtained gain in performance in comparison with the initial performance $Perf_{init}$ obtained with the default setting.

IV. Experimental Setup

A. The Use Case

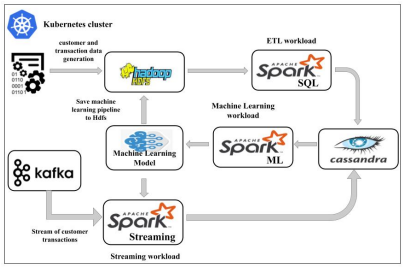

Fig. 3. The fraud detection big data pipeline.

With the emergence of payment systems and the increasing customer confidence in electronic payments, fraud detection has become a critical factor. However, detecting fraudulent transactions within seconds - so that the card provider can stop the transaction - requires scalable machine learning techniques and an architecture that can ingest, process, and analyze massive amounts of data. Fortunately, the expansion of the

open-source big data frameworks brought new possibilities and perspectives to the fraud detection field. In this paper, we used the fraud detection use case as an input application to our tuning agent, aiming at optimizing its performance as it is close to real-world use cases and combines both streaming and batch processing workloads, leveraging popular big data frameworks. This fraud detection pipeline, shown in Fig. 3, is based on an existing Github repository [1], and integrates four Big Data frameworks that can be tuned:

Kafka: is a distributed messaging system that decouples processing from data producers and stores streams of events durably.

Spark: is a unified big data processing engine, with integrated modules for streaming, SQL, machine learning, and graph processing.

Cassandra: is a NoSQL distributed database that offers high availability and fault tolerance without sacrificing performance.

Hadoop Distributed File System (HDFS): is a highly available distributed file system that stores large files and provides high throughput to access them.

As shown in Fig. 3, the fraud detection application is based on the following three main workloads:

- **The extract, transform and load workload (ETL)** responsible for reading and transforming transactions and customer data from HDFS using Spark SQL and then storing it in Cassandra.
- **The Machine Learning workload** is responsible for reading data from Cassandra, and pre-processing them, then finally training a logistic regression model using Spark ML to detect fraudulent transactions.
- **The streaming workload** is responsible for reading a stream of transactions from Kafka and checking if the transaction is a fraud, and finally storing the results in Cassandra.

In order to improve the original fraud detection pipeline and adapt it to our needs, we made the following changes: First, we integrated HDFS as a storage layer and implemented a data generator for customer and transaction data. Second, we worked on the virtualization of the entire fraud detection pipeline as a Docker container, and prepared the pipeline for deployment in Kubernetes. Finally, we replaced the Random Forest model used in the machine learning workload with Logistic Regression, as Random Forest consumes too much resources to the point that running with the default configuration became impossible.

B. The Test Scenarios

In this work, we compare the performance of the reinforcement learning algorithm TD3 for tuning the big data pipeline, against a baseline, i.e. random search.

For our TD3 agent, we evaluate the effectiveness of two different experience replay strategies, namely the classic random experience replay and the prioritized experience replay. The TD3 agent, with both replay strategies, is trained over 200 steps divided over 4 episodes of 50 steps each. The agent starts with an exploration phase over 32 steps after which the agent starts following a policy. The batch size used to train the networks is 16.

We conducted our experiments using the fraud detection big data pipeline with different input data sizes. For the first scenario, we used 5 million rows for the batch workloads, and for the second, we used 10 million rows. The streaming input rate is fixed to 200 rows/s for both scenarios.

We ran these experiments on Google Kubernetes Engine (GKE) with 5 nodes, each with 4 CPUs and 15 GB of memory.

V. Results and Discussion

The obtained results show that for the fraud detection pipeline, with a continuous action space of 46 parameters, our tuning agent achieves a better performance than random search, regardless of the memory replay strategy used during the training. In fact, for the first test scenario, with 5 Million rows as input data, the random agent improves the default performance by 25% only, raising the initial performance from about 0.14 to 0.17 with a high standard deviation of 0.06. However, the TD3 agent, for the same test scenario, and after 200 steps of training only, improves the default performance by around 60% bringing the performance to a higher threshold of 0.22 with a small standard deviation of about 0.009.

We also observed that the agent trained with prioritized experience replay performs better than the one trained with the random experience replay. In fact, the former achieves a 6% better performance than the latter. Fig. 4 shows that the TD3 agent trained with the prioritized experience replay achieves a higher performance during the training process compared to the one trained with random experience replay. This leads to a better policy at evaluation, as the aforementioned results show.

Details about the obtained results with the different tuning methods are presented in Table II, Table III and Table IV for the performance, the end-to-end latency, and the end-to-end throughput respectively.

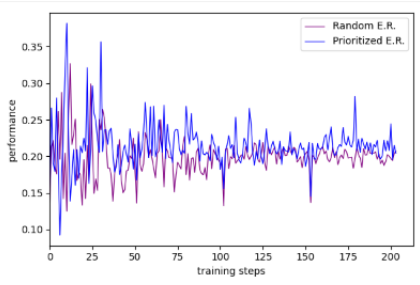

Fig. 4. TD3 training process with different replay strategies.

A similar behavior is observed with a higher input data size of 10M rows. The random agent has a performance improvement rate of 29% against 39% and 41% for the two TD3 agents with random experience replay and prioritized experience replay respectively.

[1]https://github.com/SainathDutkar/Fraud_Transaction_Monitor

TABLE II
PERFORMANCE RESULTS

Data size	Agent	avg. perf.	std.dev.	Imprv
5M rows	Default	0.14274	–	–
	Random Search	0.17204	0.06090s	25.66%
	TD3 - Random E.R	0.22321	0.0073s	56.37%
	TD3 - P.E.R	**0.22640**	0.0094s	**63.13%**
10M rows	Default	0.09762	–	–
	Random Search	0.12670	0.03424	29.79%
	TD3 - Random E.R	0.13604	0.00927	39.36%
	TD3 - P.E.R	**0.13782**	0.00554	**41.18%**

TABLE III
END-TO-END LATENCY RESULTS

Data size	Agent	avg. lat.	std.dev.	Imprv
5M rows	Default	502.999s	–	–
	Random Search	434.754s	131.902s	-15.11%
	TD3 - Random E.R	377.547s	9.076s	-24.94%
	TD3 - P.E.R	**374.110s**	10.195s	**-26.62%**
10M rows	Default	840.332s	–	–
	Random Search	759.95s	191.976s	-10.34%
	TD3 - Random E.R	731.47s	23.450s	-12.95%
	TD3 - P.E.R	**681.289s**	17.907s	**-18.92%**

Detailed results for the performance, the end-to-end latency, and the end-to-end throughput can be found in Table II, Table III and Table IV.

VI. CONCLUSION AND FUTURE WORKS

In this paper, we propose to apply an actor-critic algorithm, namely Twin Delayed Deep Deterministic Policy Gradient, TD3, to automatically tune a big data pipeline. For the purpose of this article, we applied the tuning agent to optimize the performance of a fraud detection application. The choice of such a use case was made to show the performance of TD3 on a near real world use case, keeping in mind that the pipeline includes a number of widely used tunable big data frameworks. We have shown that TD3 improves the performance by 63% compared to the default configuration with only 200 steps despite the size of the continuous action space.

The experimental results also showed that prioritized experience replay improves the performance of the agent compared to the conventional random experience replay. However, it should be mentioned that the difference in performance between agents trained with the two strategies is not significant. In our future work, we suggest exploring and comparing other machine learning solutions to further improve the tuning performance of big data pipelines. We are also interested in meta-reinforcement learning to test its ability to adapt to workload variation on the same pipeline. In fact, it would be interesting to train a single model on different tasks and validate it on a new, previously unknown task. The agent will then recommend the best configuration based on its knowledge of previously tuned applications.

TABLE IV
END-TO-END THROUGHPUT RESULTS

Data size	Agent	avg. tput	std.dev.	Imprv
5M rows	Default	71.385	–	–
	Random Search	77.038	21.608	9.79%
	TD3 - Random E.R	97.094	1.345	25.71%
	TD3 - P.E.R	**97.61**	1.779	**27.25%**
10M rows	Default	81.661	–	–
	Random Search	92.602	21.608	13.80%
	TD3 - Random E.R	99.733	2.850	18.07%
	TD3 - P.E.R	**100.478**	2.601	**18.82%**

REFERENCES

[1] P. A. Ivanenko, A. Y. Doroshenko, and K. A. Zhereb, "TuningGenie: Auto-Tuning Framework Based on Rewriting Rules," in *Information and Communication Technologies in Education, Research, and Industrial Applications*, V. Ermolayev, H. C. Mayr, M. Nikitchenko, A. Spivakovsky, and G. Zholtkevych, Eds. Cham: Springer International Publishing, 2014, pp. 139–158.

[2] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do Not Blame Users for Misconfigurations," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 244–259.

[3] R. Suda, K. Naono, K. Teranishi, and J. Cavazos, *Software Automatic Tuning: Concepts and State-of-the-Art Results*. New York, NY: Springer New York, 2010, pp. 3–15.

[4] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A Self-tuning System for Big Data Analytics," in *In CIDR*, 2011, pp. 261–272.

[5] C. Liu, D. Zeng, H. Yao, C. Hu, X. Yan, and Y. Fan, "MR-COF: A Genetic MapReduce Configuration Optimization Framework," in *Algorithms and Architectures for Parallel Processing*, G. Wang, A. Zomaya, G. Martinez, and K. Li, Eds. Cham: Springer International Publishing, 2015, pp. 344–357.

[6] J. Shi, J. Zou, J. Lu, Z. Cao, S. Li, and C. Wang, "MRTuner: A Toolkit to Enable Holistic Optimization for Mapreduce Jobs," *Proc. VLDB Endow.*, vol. 7, no. 13, pp. 1319–1330, aug 2014.

[7] N. Mishra, H. Zhang, J. D. Lafferty, and H. Hoffmann, "A probabilistic graphical model-based approach for minimizing energy under performance constraints," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, p. 267–281, Mar. 2015.

[8] S. Kadirvel and J. A. B. Fortes, "Grey-Box Approach for Performance Prediction in Map-Reduce Based Platforms," in *2012 21st International Conference on Computer Communications and Networks (ICCCN)*, jul 2012, pp. 1–9.

[9] K. Wang, M. Maifi, and H. Khan, "Performance Prediction for Apache Spark Platform," 2015.

[10] M. Cardosa, P. Narang, A. Chandra, H. Pucha, and A. Singh, "STEAMEngine: Driving MapReduce provisioning in the cloud," in *2011 18th International Conference on High Performance Computing*, dec 2011, pp. 1–10.

[11] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, "A simulation approach to evaluating design decisions in MapReduce setups," in *2009 IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems*, 2009, pp. 1–11.

[12] 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 2017 - Proceedings of the 2017 Symposium on Cloud Computing*. Association for Computing Machinery, Inc, sep 2017, pp. 338–350.

[13] G. Liao, K. Datta, and T. L. Willke, "Gunther: Search-Based Auto-Tuning of MapReduce," in *Euro-Par 2013 Parallel Processing*, F. Wolf, B. Mohr, and D. an Mey, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 406–419.

[14] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller, "MRONLINE - MapReduce Online Performance Tuning," *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing - HPDC '14*, 2014.

[15] S. Kumar, S. Padakandla, L. Chandrashekar, P. Parihar, K. Gopinath, and S. Bhatnagar, "Scalable Performance Tuning of Hadoop MapReduce: A Noisy Gradient Approach," in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, jun 2017, pp. 375–382.

[16] S. Duan, V. Thummala, and S. Babu, "Tuning database configuration parameters with ituned," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 1246–1257, 2009.

[17] H. Wang, S. Rafatirad, and H. Homayoun, "A+ Tuning: Architecture+Application Auto-Tuning for In-Memory Data-Processing Frameworks," in *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, dec 2019, pp. 163–166.

[18] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li, "An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 415–432.

[19] C. Bitsakos, I. Konstantinou, and N. Koziris, "DERP: A deep reinforcement learning cloud system for elastic resource provisioning," in *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, vol. 2018-Decem, 2018, pp. 21–29.

[20] G. Li, X. Zhou, S. Li, and B. Gao, "QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 2118–2130, aug 2019.

[21] H. Du, P. Han, Q. Xiang, and S. Huang, "Monkeyking: Adaptive parameter tuning on big data platforms with deep reinforcement learning," *Big Data*, vol. 8, no. 4, pp. 270–290, 2020.

[22] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[23] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[24] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2016.

[25] 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," 2019.

[26] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," 2018.

[27] H. V. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *ArXiv*, vol. abs/1509.06461, 2016.