Self-tunable DBMS Replication with Reinforcement Learning

Luís Ferreira, Fábio Coelho, José Pereira

Self-Tunable DBMS Replication with Reinforcement Learning

Luís Ferreira[1][0000−0003−3364−3670], Fábio Coelho ✉[1][0000−0002−0188−6400], and José Pereira[1][0000−0002−3341−9217]

INESC TEC & Universidade do Minho, Braga, Portugal
{luis.m.ferreira,fabio.a.coelho}@inesctec.pt,jop@di.uminho.pt

Abstract. Fault-tolerance is a core feature in distributed database systems, particularly the ones deployed in cloud environments. The dependability of these systems often relies in middleware components that abstract the DBMS logic from the replication itself. The highly configurable nature of these systems makes their throughput very dependent on the correct tuning for a given workload. Given the high complexity involved, machine learning techniques are often considered to guide the tuning process and decompose the relations established between tuning variables. This paper presents a machine learning mechanism based on reinforcement learning that attaches to a hybrid replication middleware connected to a DBMS to dynamically live-tune the configuration of the middleware according to the workload being processed. Along with the vision for the system, we present a study conducted over a prototype of the self-tuned replication middleware, showcasing the achieved performance improvements and showing that we were able to achieve an improvement of 370.99% on some of the considered metrics.

Keywords: Reinforcement learning, dependability, replication

1 Introduction

Distributed systems, namely scalable cloud-based Database Management Systems (DBMS), encompass a growing number of tunable criteria that may deeply affect the system's performance [5]. This is particularly true for replicated DBMS as replication capabilities are often tightly connected with the overall system design, and the lack of proper tuning may impact on the system dependability, degrading quality of service. Moreover, the inner characteristics of each workload also directly affect how the DBMS engine performs.

The number and type of adjustable criteria available in each DBMS [4,18,16] varies, but generally, they allow to configure a common set of properties, such as memory space available, the number of parallel workers allowed, together with the dimension of execution pools for specific tasks. In what regards replication, usually systems allow to tune the type of replication mechanism considered, the number of active instances, the acknowledgement of delays admissible and how far can a given replica drift on data consistency before the system's dependability is compromised.

The perceived performance of the system is usually measured through the number of executed operations in a time frame, which is often collected as an exogenous property provided by external benchmarking tools, and not as an intrinsic system property. Ultimately, achieving the best possible configuration is the result of a multi-stage process where trial and error plays an important role, attributing to the database administrator (DBA) the tuning responsibility and centring on him/her the intuition of former adjustments and their consequences. Even when considering this approach, the DBA is faced with a daunting task as the recommendations instruct configurations to be changed one-a-time, but it is clear that tunable configurations are not independent [5].

The rapid expansion of autonomous and machine learning techniques is currently pushing these approaches to be considered in the optimization of systems, namely for pattern discovery and recognition, and for self-tuning of systems.

Classical supervised learning approaches split the process into distinct stages, encompassing a period strictly for learning based on previous feedback and a second one for making predictions. The decoupling of these stages is restrictive as they are disjoint, and typically incorporating new data patterns into the model implies a new training period. The use of techniques that merge both stages, learning and making predictions in unison *e.g.,* Reinforcement Learning (RL) [20] promises to overcome that limitation.

This paper presents a self-tunable mechanism based on Reinforcement Learning, to configure and adjust in real time a hybrid replication middleware, paired together with a DBMS system. In a nutshell, when requests arrive at the replication middleware, they are split into shards, *i.e.,* a set of logical partitions based on the workload, which are then assigned to distinct DBMS replicas. The system is able to probe the middleware configuration and tune it to an ideal configuration in real-time, without having to incur in the cost of restarting the middleware or the underlying DBMS. Moreover, it does so in a dynamic manner, the configuration is constantly adjusted to reflect the best values for the current workload.

We deployed and evaluated the system, considering the TPC-C benchmark to inject a workload into PostgreSQL [17], the underlying DBMS system attached. Results show that for some of the metrics considered, the gains were of 370.99%.

The rest of this paper is organized as follows: Section 2 provides the core concepts and motivation for the system, while Section 3 goes over the design of the architecture. The use of reinforcement learning is detailed in subsection 3.3 and the system's evaluation in subsection 3.5. Finally, related work is presented in Section 4 and Section 5 concludes this paper, highlighting the major takeaways and future work.

2 Motivation

Nowadays, there is a plethora of DBMS, each focusing on a particular workload type, namely: OnLine Transactional Processing (OLTP), OnLine Analytical Processing (OLAP) and even newer Hybrid Transactional and Analytical Processing (HTAP) workloads. Even though there are guidelines on how to assess and tune a DBMS, the configurations are not universal and each vendor decides on which tunable configurations to expose. Moreover, workload types are very distinct to allow a common configuration across systems. This is so as optimizing an OLTP targeted operation would intrinsically degrade OLAP performance and vice-versa [6]. Therefore, as the type of configurations available for tuning is increasing, and, most importantly, the fact that the challenges associated with a given workload type are different for each vendor, renders the DBA with most of the know-how and responsibility to assess and fine tune a workload in a particular DBMS.

This is particularly true for replication mechanisms and for the provision of fault-tolerance and high-availability, as such features are usually deeply connected with the way each DBMS handles data, particularly in OLTP systems. Database systems typically provide fault-tolerance capabilities through replication at the cost of lowering the throughput, where the algorithms are intricately deployed as part of the DBMS's kernel. When pairing a DBMS engine with the provision of fault-tolerance guarantees through external middleware systems, the number of tunable configurations considered increases. The advantages that come with the decoupling such as modularity and pluggability, come at the expense of higher complexity and a possible detachment between the design consequences of the replication middleware and their impacts on the underlying DBMS considered. Moreover, the logical detachment is typically achieved through standard interfaces such as the Java DataBase Connector (JDBC), which also imposes new concerns.

As to reduce the complexity and automate the decision and tuning process associated, particularly with the provision of fault-tolerance, we envision a system architecture aided through the use of machine forecasting, via reinforcement learning techniques.

Therefore, successfully joining the configurability and detachment of a replication middleware, with the capabilities of RL techniques to self-adjust the system, would reduce the key features undermining the current solution.

3 System Design

On the basis of the previously described vision, the proposed architecture is depicted in Figure 1.

3.1 Overview

The system is abstracted in three distinct components, namely: *(i)* the middleware replica nodes holding the metric Monitor and the Update Handler, *(ii)* the reinforcement learning system holding the metric Monitor Handler, the Update Dispatcher and the Reinforcement Learning Agent itself and *(iii)* the underlying DBMS. The replication middleware is built around the Apache Bookkeeper Distributed Log (DL) [1].

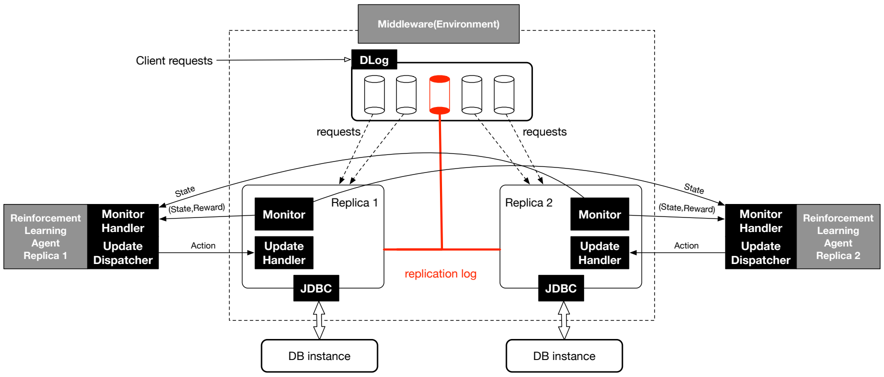


Fig. 1: System architecture.

In a nutshell, the replication middleware receives incoming JDBC requests from clients, which are then fed into a group of distributed log instances. The distributed log is an efficient and high-performance solution to enable replication and replica communication. Its interface considers two roles in its architecture, namely: writers and readers; holding the responsibility to push and read records from the incoming events received through the log stream, *i.e.,* incoming database requests. The requests undergo a sharding procedure [8], splitting them in slots according to a configurable number of DataBase (DB) instances and mapping each distributed log instance to a group of shards. The replica nodes execute the client requests assigned to them, while also writing the resulting state changes to the replication log. They also receive the replicated state changes from other replicas and update their records accordingly. The replication middleware is based on the active replication protocol [24], with a semi-active variant [13]. On one hand, as shards are assigned to a replica, that replica becomes the primary, handling requests for that shard. On the other hand, the same replica acts as backup, incorporating updates of other primary replicas in charge of other shards.

The Reinforcement Learning system is built from a set of subcomponents: the Monitor and Update Handler, that act as probes in each replica, and, the Monitor Handler, the Update Dispatcher and the Learning Agent. The architecture comprehends an instance for each one of these components per replica.

The Monitor probes and aggregates metrics (section 3.3) about each replica that are afterwards pushed to the Monitor Handler. The results are then fed into the Reinforcement Learning Agent that adjusts a base configuration and instructs replicas to modify their configurations via the Update Dispatcher and the replica's local Update Handler probe. Changes are applied on-line, without having to restart the middleware or the underlying DB instances.

In this paper we considered for adjustment all the tuning variables that are made available by the middleware, as depicted in Table 1. The set of possible actions for the reinforcement learner is composed by incremental changes to these variables. Also, it should be noted that each replica has an independent reinforcement learning agent controlling its tuning variables, which can be deployed in a different machine. This allows us to add new replicas to the cluster without having to retrain the model for the already existing replicas. Also, since different replicas process different shards, the workload may vary between them. By tuning each one individually, we get a configuration that is optimized for the workload of each replica.

Table 1: Adjustable configurations considered for on-line adjustment.

Configuration	Description
db.pool	Size of the connection pool
dlog.reader.threads	Number of worker threads
db.writeset.transactions	Max batch size
db.writeset.delay	Delay between writes to replication log

3.2 Components

In order to feed the Reinforcement Learning mechanism, the design comprehends a set of custom components to observe and collect metrics, but also, to trigger the required configuration changes. The components are split into two groups, the ones pertaining to the replica, the probes, and the ones that pertain to the reinforcement learning agent.

Monitor The monitor component is deployed in each replica of the replication middleware. It probes a set of local properties that are then fed into the learning mechanism. The cornerstone of this component is a publish-subscribe queueing service [9], that periodically pushes readings into the Monitor Handler component. In practice, this component feeds the reinforcement learning algorithm with state and reward updates. Updates are sent every 15 seconds.

Update Handler The Update Handler component is deployed in each replica of the replication middleware. It receives asynchronous actions from the Update

Dispatcher and applies them directly in each node of the middleware (our environment), allowing a dynamic change without having to restart the middleware or the DB engine.

Monitor Handler The Monitor Handler component is deployed outside the replica nodes, in each of the Reinforcement Learning Agents. It collects the metrics sent by the Monitor agents in each replica by subscribing their updates through the publish-subscribe service. This data will be used to determine the current state of the environment and the reward associated with an action.

Update Dispatcher The Update dispatcher component is also deployed on each of the Reinforcement Learning Agents. It considers a set of actions dictated by the RL component and triggers the Update Handlers to impose the changes into the new configurations.

3.3 Reinforcement Learning Agent

Each replica is controlled by a distinct Reinforcement Learning agent. The RL component is responsible for analysing data arriving from the Monitor via the Monitor Handler. At each step, the RL algorithm characterizes incoming raw data from the Monitor component into the *state* to be fed to the RL algorithm, which will be translated by the agent's policy to an *action*. That action is then applied to the environment, the middleware replica. The reward derived from an action is also obtained from the data retrieved by the Monitor Handler.

The tuning of the configuration variables is done dynamically, meaning that configuration values are constantly being fine-tuned, in response to changes in the workload.

The search space associated to these strategies is characterized by a combinatorial set built from all possible states and actions. The data that is collected, the state, is characterized by the variables identified in Table 2, a set of discrete variables, with a large search space associated. The algorithm chosen was the *Deep Q-learning* algorithm, as it can accommodate larger search spaces. This mechanism incorporates a complex neural network on top of the *Q-learning* algorithm [23]. Given this choice, the action space was sampled into a subset of 10 possible choices, depicted in Table(3).

States The states taken into account by the RL method are a composition of the current values of the variables depicted in Table 1, which are then complemented by a set of metrics that characterize the overall performance of the system in each replica. These metrics represent an average value over the period comprehended between two consecutive metric readings. The complete set of elements that compose the state of our Reinforcement Learning model are depicted in Table 2.

The metrics allow to establish a relationship between the number of requests received in a given replica and the number of requests executed in that same replica. Moreover, the number of received and executed transactions on other replicas of the system (which will not be optimised) are also part of the state. This allows to establish a ratio between the number of replicated requests executed and the total number of requests that other replicas have already sent to the distributed log for persistence. Thus, it allows the system to know whether or not a lower throughput of replicated transactions is due to a lack of performance from the replica itself or from the replica that is sending the updates.

Table 2: Set of elements that compose the state in the RL process.

Configuration	Description
db.pool	Current size of the connection pool.
dlog.reader.threads	Current number of worker threads.
db.writeset.transactions	Current max batch size.
db.writeset.delay	Current delay between writes to replication log.
ClientRequests	Executed transactions.
Txn Written	Transactions' (Txn) state updates sent to replication log.
Replicated Txn	Replicated transactions' state updates applied to the DBMS.
ClientRequests	Nr. of new client requests for this replica.
rep_Txn in DL	Transactions sent to replication log by other replicas.

The current values for the variables that are being adjusted is made part of the state so that relations between the variables' values and between them and the collected metrics can be established by the neural network, since the possible actions, the outputs of our neural network, do not reflect actual values for the variables, but rather incremental changes to them.

Actions The actions decided by the neural network build the possible changes that can be made on each step to the environment. While using Deep Q-Learning, variables were sampled in the form of incremental changes. The possible set of actions is depicted in Table 3. In order to prevent system crashes, an upper and lower limit was set for each tuning variable. The increment and decrement values were chosen by trial and error. The objective was to choose increments that wouldn't be so small that they wouldn't have a significant effect on the performance of the system, but not so large that the number of possible values for each variable became very low (taking into account the boundaries set).
In summary, with each step of the algorithm, only one of the above mentioned actions is executed. This means that only one variable is changed in each step, and its value is only changed by a small amount, as described in Table 3.
The configuration variable *db.writeset.transactions* can be set to a special value, 0. When this value is set, the limit on how many transactions can be written on each batch is removed.

Table 3: Actions considered in the RL process.

Actions	Description
No action	Nothing is changed.
Increment db.pool	Incremented by 1.
Decrement db.pool	Decremented by 1.
Increment dlog.reader.threads	Incremented by 1.
Decrement dlog.reader.threads	Decremented by 1.
Increment db.writeset.transactions	Increment by 100
Decrement db.writeset.transactions	Decrement by 100
Set db.writeset.transactions special	Set to 0
Increment db.writeset.delay	Increment by 100
Decrement db.writeset.delay	Decrement by 100

Reward Function As the environment being considered is bounded to database replication and overall database execution, the impact can be directly associated with the overall throughput. A higher throughput represents a better outcome. Consequently, the reward function is associated with the throughput, being defined as the sum of all latest metric averages that refer to replica throughput. Since all the averages are in transactions per second, there is no need for any normalization or transformation to be applied to the values. In this case, all reward components have the same weight towards the computed reward.

$$reward = ClientRequests + TxnWritten + ReplicatedTxn \tag{1}$$

3.4 Reinforcement Learning Mechanism

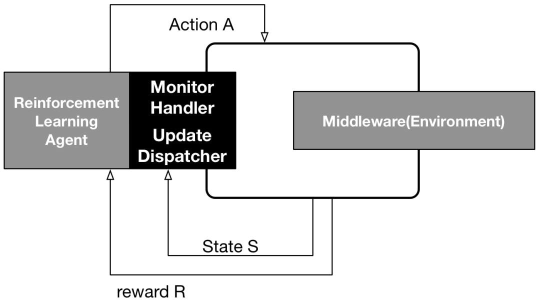


Fig. 2: Reinforcement Learning in the environment

The mechanism proposed considers RL based on Deep Q-Learning. Within a RL approach, the Agent monitors a certain Environment. For a given State the agent decides on an Action, by means of its Policy. A Policy maps States to Actions. The objective of the agent is to find the best possible Policy, the one that maximizes the considered Reward. Reward values are calculated after an Action is applied to the Environment using the Reward Function considered. The system is depicted in Figure 2.

The Deep Q-Learning algorithm sets an evolution over Q-Learning. The latter establishes a table that defines the agent's policy. It cross-matches the possible actions and states, attributing to each combination a q-value.

At the beginning of the training process, all q-values are equal to zero. The policy defines that for each state we should choose the action with the highest q-value, corresponding to the biggest reward. However, always choosing the action with the highest q-value could prevent the discovery of alternative plausible actions, which could become exacerbated as the number of states increases. So, a small percentage of actions are chosen at random.

Introducing a neural network in place of a state table, enables Deep Q-Learning to consider a larger combinatorial set of states. Each output node represents a possible action and the value calculated by the network for that node will be the q-value of the action. The neural network considered is depicted in Figure 3. It holds two hidden layers of 16 neurons each. In each step, the selected action will be the one with the highest q-value. Rewards are used to adjust the weights of the neural network by back-propagation. According to Figure 3, the chosen action would be incrementing *dlog.reader.threads*, because it's the one with the highest q-value, in the step depicted.

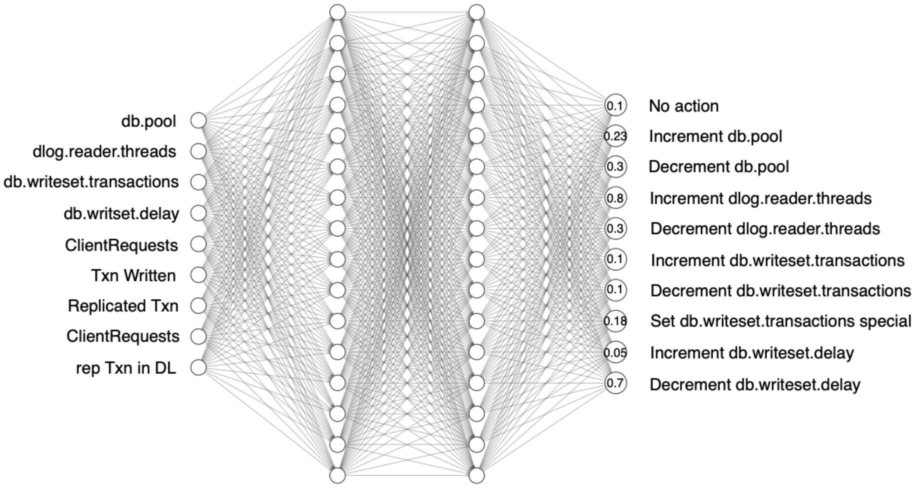

db.pool
dlog.reader.threads
db.writeset.transactions
db.writset.delay
ClientRequests
Txn Written
Replicated Txn
ClientRequests
rep Txn in DL
0.1
No action
0.23
Increment db.pool
0.3
Decrement db.pool
0.8
Increment dlog.reader.threads
0.3
Decrement dlog.reader.threads
0.1
Increment db.writeset.transactions
0.1
Decrement db.writeset.transactions
0.18
Set db.writeset.transactions special
0.05
Increment db.writeset.delay
0.7
Decrement db.writeset.delay

Fig. 3: Neural Network used in the Reinforcement Learning agents. Computed q-values are depicted on the right hand side, paired with the respective action.

3.5 Preliminary Evaluation

The evaluation of the system was built considering the TPC-C benchmark, designed specifically for the evaluation of OLTP database systems. The TPC-C specification models a real-world scenario where a company, comprised of several warehouses and districts, processes orders placed by clients. The workload is defined over 9 tables operated by a transaction mix comprised of five different transactions, namely: New Order, Payment, Order-Status, Delivery and Stock-Level. Each transaction is composed of several read and update operations, where 92% are update operations, which characterizes this as a write heavy workload.

Table 4: Configuration tuned with reinforcement learning from base configuration. Baseline and cycle results in transactions per second (Txn/sec).

Metric	Baseline	RL#1	Gain	RL#2	Gain
ClientRequests-R1	80.80	94.07	**+16.42%**	91.87	**+13.70%**
Replicated Txn-R1	35.24	55.56	**+57.64%**	55.00	**+56.05%**
Txn Written-R1	27.57	61.73	**+123.92%**	91.87	**+233.25%**
ClientRequests-R2	178.20	129.51	-27.32%	157.82	-11.44%
Replicated Txn-R2	27.16	61.75	**+127.40%**	91.56	**+237.18%**
Txn Written-R2	31.86	129.51	**+306.44%**	150.08	**+370.99%**
Avg Reward-R1	143.61	211.35	**+47.17%**	238.74	**+66.24%**
Avg Reward-R2	237.22	320.78	**+35.22%**	399.46	**+68.39%**

Metric	Baseline	RL#4	Gain	RL#6	Gain
ClientRequests-R1	80.80	99.96	**+23.71%**	86.04	**+6.49%**
Replicated Txn-R1	35.24	52.57	**+49.15%**	52.09	**+47.79%**
Txn Written-R1	27.57	99.96	**+262.59%**	86.04	**+212.11%**
ClientRequests-R2	178.20	142.30	-20.15%	207.00	**+16.16%**
Replicated Txn-R2	27.16	99.96	**+268.09%**	68.21	**+151.16%**
Txn Written-R2	31.86	142.30	**+346.57%**	111.55	**+250.09%**
Avg Reward-R1	143.61	252.48	**+75.81%**	224.17	**+56.09%**
Avg Reward-R2	237.22	384.55	**+62.11%**	386.75	**+63.03%**

Metric	Baseline	RL#8	Gain	RL#10	Gain
ClientRequests-R1	80.80	111.92	**+38.52%**	112.95	**+39.79%**
Replicated Txn-R1	35.24	30.45	**-13.59%**	69.08	**+96.01%**
Txn Written-R1	27.57	75.46	**+173.74%**	112.95	**+309.73%**
ClientRequests-R2	178.20	220.57	**+23.77%**	205.47	**+15.30%**
Replicated Txn-R2	27.16	68.20	**+151.15%**	98.25	**+261.81%**
Txn Written-R2	31.86	96.73	**+203.57%**	94.26	**+195.82%**
Avg Reward-R1	143.61	217.84	**+51.69%**	294.99	**+105.41%**
Avg Reward-R2	237.22	385.50	**+62.51%**	397.99	**+67.77%**

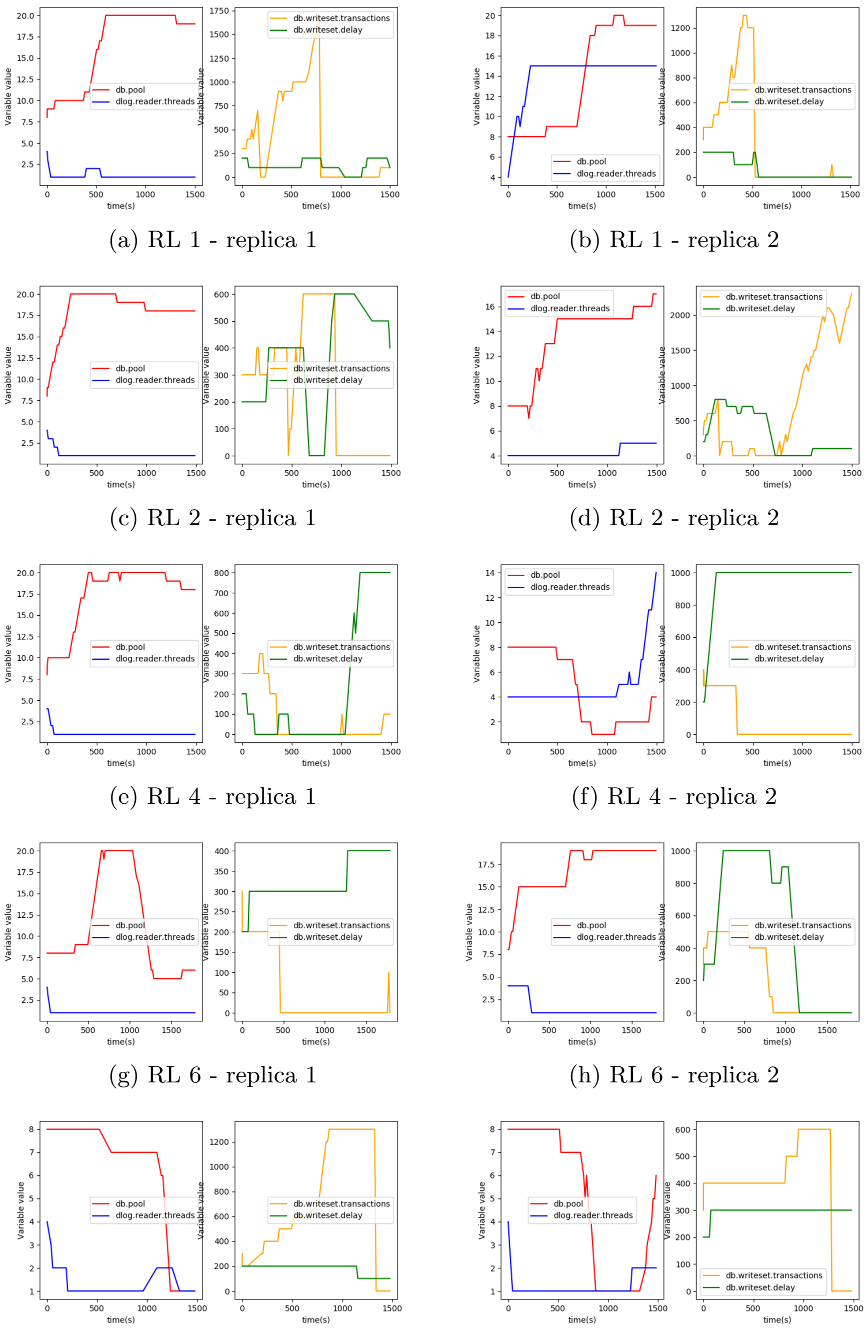
Setup TPC-C was set up in a configuration comprising 150 warehouses with a load of 50 client connections per warehouse. Moreover, the middleware replication component was set up to accommodate 25 warehouses per distributed log instance. Tests were conducted on a local server built from a Ryzen 3700 CPU, 16GB of RAM and 2 SSDs (with one of them being NVME), hosting all services. Replica 1 (R1) and two distributed log bookies used the NVME driver, while replica 2 (R2) and one other bookie used the other SSD drive.

Overall, the evaluation was achieved by running the TPC-C benchmark and while it was sending transactions to be committed on the underlying database through the replication middleware, the reinforcement learning agent was deployed on a separate machine.
A total of 10 learning cycles were run, all starting from the same baseline configuration. The initial baseline configuration was built from the researchers assumptions of a reasonable configuration for the system being used. The initial baseline configuration is the same for all learning cycles so that the initial state of the middleware doesn't differ between them.
The first 5 cycles were adjusted for a faster learning process, leaning more towards exploration. This series of cycles was comprised by 100 steps each, updating the Neural Network's weights every 15 steps, and with a probability for the action taken being chosen randomly of 20%.
On the final 5 cycles, weights were updated every 20 steps and actions were chosen at random only 10% of the time. The number of steps was also increased to 120 steps. The number of steps was chosen so that the reinforcement learning agent is active for about the same time that the benchmark takes to conclude. Each step involves taking one of the actions described in Table 3. The final weights of the neural network on each cycle were transferred to the next, in order to incrementally refine the Neural Network used by the Reinforcement Learning agent.

It is worth noting that the learning mechanism benefits from environment state updates within short intervals, which would otherwise induce large delays in the learning process, hence the custom monitoring system implemented.

Results Table 4 depicts the results for a subset of the learning cycles of the reinforcement learning agent. The average results reported are the average results for each evaluation cycle. The results show the impact on both deployed replicas, depicted as R1 and R2. We can see that on all learning cycles, the performance was better than for the baseline case. We can also see that the average reward value tends to increase in the case of R1, while in the case of R2, the maximum gain seems to have been achieved, at around 68%.

The actions that were taken in each RL adjustment are depicted in Figure 4 and 5. The figures depict 6 of the 10 cycles, detailing the evolution of each considered configuration.



(a) RL 1 - replica 1

(b) RL 1 - replica 2

(c) RL 2 - replica 1

(d) RL 2 - replica 2

(e) RL 4 - replica 1

(f) RL 4 - replica 2

(g) RL 6 - replica 1

(h) RL 6 - replica 2

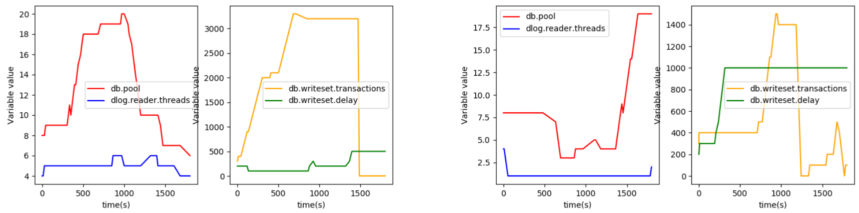
(i) RL 8 - replica 1

(j) RL 8 - replica 2

Fig. 4: Evolution of actions taken on configuration variables during benchmarking, Cycle 1,2,4,6 and 8.

It is possible to observe that the pattern for each configuration variable evolved over the learning cycles, and that those patterns differ for each replica.

It's also worth noting that there is a distributed component, and therefore, a dependency between. The value for the Replicated Txn metric (Table 2) of one replica can only be as high as the value for the TxnWritten metric of the other replica. This means that the performance of a replica is partly influenced by the performance of the other.



(a) RL 10 - replica 1

(b) RL 10 - replica 2

Fig. 5: Evolution of actions taken on configuration variables during benchmarking. Cycle 10.

Moreover, a deeper look into the results depicted, allows complementary conclusions, namely the fact that across all observed cycles in both replicas, the *db.writeset.transactions* is set to 0 at some point. This implies that the replicas were running with no limits for batching operations. Moreover, the *dlog.reader.threads*, which guides the thread pool associated with the distributed log read handler (in charge of acquiring data) was mostly reset to minimum values, which highlights the intense write profile of the considered benchmark.

Table 5 overviews the final results, extracted from Table 4. Overall, comparing the baseline results to the last tuning cycle, the state metrics directly associated with the replication mechanism registered a significant improvement, an order of magnitude higher for some of the state parameters.

Table 5: Results overview.

Metric	Baseline(Txn/sec)	RL#10(Txn/sec)	Gain
ClientRequests-R1	80.80	112.95	**+39.79%**
Replicated Txn-R1	35.24	69.08	**+96.01%**
Txn Written-R1	27.57	112.95	**+309.73%**
ClientRequests-R2	178.20	205.47	**+15.30%**
Replicated Txn-R2	27.16	98.25	**+261.81%**
Txn Written-R2	31.86	94.26	**+195.82%**

This scenario holds a limited number of tuning variables, so a neural network with two 16 hidden layers was enough to capture the complexity of the problem. Furthermore, our state contains information limited to the replication middleware action, holding no variables associated with the underlying hardware, for instance. These characteristics allowed us to train our model in a very short time, but deny the possibility of applying the trained model from one machine to another. We defer to future work the problem of inter-machine compatibility of the models.

4 Related Work

Prominent related work is generally focused in providing optimisations towards the physical deployment and data layouts of database systems. The linkage with the physical data layout allow self-adjustment systems to expedite index [22] creation, data partitions [3] or materialized views [2]. However, these approaches are unable to capture the details behind the DBMS, where the adjustment of specific tuning requirements impact on the DBMS internals. Most database products like IBM BD2, Microsoft SQL Server or MySQL include tools for tuning their respective systems in respect to the server performance [15,4]. Broadly speaking, these systems test alternative configurations in off-production deployments and assess their outcome. However, this solutions are bound to the discretion of the DBA in selecting the strategies to apply and what subsequent actions are taken in the DBMS.

The set of tools described also (indirectly) cover tunable criteria that strictly concerns the adjustment of the replication techniques, as part of the tunable configuration variables that are made available in each system. However, the consideration of tunable variables is generally absent [5], although some systems specialize in selecting the most meaningful configurations [19]. Even so, this is usually an independent process.

Although not a novelty in the machine and autonomous learning interest groups, Reinforcement Learning strategies are currently trendy techniques to be applied in distinct realms of computation. This class of techniques encompass an agent that tries to acquire state changes in their respective environment while aiming to maximize a balance of long-term and short-term return of a reward function [7]. Specifically in the field of database systems, such strategies have started to be applied in the internals of DBMSs to aid query optimisers to outperform baseline optimisation strategies for join ordering [21] and overall query planning [11]. The outcome is usually a sequential set of decisions based on Markov Decision Chains [12,14]. The considered RL technique does not change the optimisation criterion, but rather the how the exploration process is conducted.

These techniques enable the agent to search for actions during a *learning stage* where it deliberately takes actions to learn how the environment responds, or to come closer to the optimisation goal. Moreover, these techniques are com-

monly powered through heuristic-based exploration methods, built on statistics collected from the environment.
More recently, RL based solutions specific to single instance database self configuration have been presented, they leverage either metrics made available by the database [25], or information from the DBMS's query optimizer and the type of queries that compose the workload of the database [10]. In this paper we try to demonstrate that this is also possible at the level of the replication middleware. Moreover, we introduce the idea of dynamic configuration, meaning that tuning variables are seen as constantly changing, adapting their values in real time to changes in the workload.

We also show that although Deep Q-Learning can't work with continuous action spaces without them being sampled, contrary to some more recent algorithms like Deep Deterministic Policy Gradient (DDPG), it still manages to achieve very good results. The use of Deep Q-Learning versus DDPG might be advantageous in resource limited contexts, since it uses only one neural network, instead of two (one for the actor and one for the critic).

5 Conclusion

This paper details a reinforcement learning mechanism that attaches to a JDBC replication middleware, optimising its tunable configurations. An architecture was introduced, showcasing the components that are part of the design. Afterwards, we test a prototype of the system and evaluate its performance, relying on the industry standard TPC-C Benchmark as a workload injector.

The main focus of this paper shows that it is possible to consider self-learning mechanisms that are able to tune a replication middleware system, as so far, optimisation efforts consider the database engines as a whole, and do not focus in specific internal components such as the replication controllers. Moreover, they do not consider external pluggable dependability mechanisms.

The evaluation allowed us to confirm the effectiveness of these techniques, but also, the major impact that adjustable tuning variables have on the overall performance of the system. The results validate the approach, highlighting maximum improvements of around 370.99% for some of the considered metrics of the replication middleware.

Future work will guide us to improve the learning mechanism to go beyond the number of tunable criteria considered, and perhaps to recognize which ones may have more impact by analysing the system's performance.

Acknowledgements

The authors would like to thank Claudio Mezzina and the anonymous reviews for their helpful comments. The research leading to these results has received funding from the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 731218.

References

1. Apache distributed log. http://bookkeeper.apache.org/distributedlog/ (2018), [Online; accessed 19-July-2019]
2. Agrawal, S., Chaudhuri, S., Narasayya, V.R.: Automated selection of materialized views and indexes in sql databases. In: VLDB. vol. 2000, pp. 496–505 (2000)
3. Curino, C., Jones, E., Zhang, Y., Madden, S.: Schism: a workload-driven approach to database replication and partitioning. Proceedings of the VLDB Endowment **3**(1-2), 48–57 (2010)
4. Dias, K., Ramacher, M., Shaft, U., Venkataramani, V., Wood, G.: Automatic performance diagnosis and tuning in oracle. In: CIDR. pp. 84–94. CIDR (2005)
5. Duan, S., Thummala, V., Babu, S.: Tuning database configuration parameters with ituned. Proceedings of the VLDB Endowment **2**(1), 1246–1257 (2009)
6. French, C.D.: ”one size fits all” database architectures do not work for dss. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data. pp. 449–450. SIGMOD ’95, ACM, New York, NY, USA (1995). https://doi.org/10.1145/223784.223871
7. Garcıa, J., Fernández, F.: A comprehensive survey on safe reinforcement learning. Journal of Machine Learning Research **16**(1), 1437–1480 (2015)
8. George, L.: HBase: the definitive guide: random access to your planet-size data. ” O’Reilly Media, Inc.” (2011)
9. Hintjens, P.: ZeroMQ: messaging for many applications. ” O’Reilly Media, Inc.” (2013)
10. Li, G., Zhou, X., Li, S., Gao, B.: Qtune: A query-aware database tuning system with deep reinforcement learning. Proceedings of the VLDB Endowment **12**(12), 2118–2130 (2019)
11. Marcus, R., Papaemmanouil, O.: Deep reinforcement learning for join order enumeration. In: Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management. pp. 3:1–3:4. aiDM’18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3211954.3211957, http://doi.acm.org/10.1145/3211954.3211957
12. Morff, A.R., Paz, D.R., Hing, M.M., González, L.M.G.: A reinforcement learning solution for allocating replicated fragments in a distributed database. Computación y Sistemas **11**(2), 117–128 (2007)
13. Powell, D.: Delta-4: a generic architecture for dependable distributed computing, vol. 1. Springer Science & Business Media (2012)
14. Puterman, M.L.: Markov Decision Processes.: Discrete Stochastic Dynamic Programming. John Wiley & Sons (2014)
15. Schiefer, K.B., Valentin, G.: Db2 universal database performance tuning. IEEE Data Eng. Bull. **22**(2), 12–19 (1999)
16. Schnaitter, K., Abiteboul, S., Milo, T., Polyzotis, N.: Colt: Continuous online tuning. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data. pp. 793–795. SIGMOD ’06, ACM, New York, NY, USA (2006). https://doi.org/10.1145/1142473.1142592, http://doi.acm.org/10.1145/1142473.1142592
17. Stonebraker, M., Rowe, L.A.: The design of Postgres, vol. 15. ACM (1986)
18. Storm, A.J., Garcia-Arellano, C., Lightstone, S.S., Diao, Y., Surendra, M.: Adaptive self-tuning memory in db2. In: Proceedings of the 32nd international conference on Very large data bases. pp. 1081–1092. VLDB Endowment (2006)

19. Sullivan, D.G., Seltzer, M.I., Pfeffer, A.: Using probabilistic reasoning to automate software tuning, vol. 32. ACM (2004)
20. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)
21. Trummer, I., Moseley, S., Maram, D., Jo, S., Antonakakis, J.: Skinnerdb: Regret-bounded query evaluation via reinforcement learning. Proc. VLDB Endow. **11**(12), 2074–2077 (Aug 2018). https://doi.org/10.14778/3229863.3236263, https://doi.org/10.14778/3229863.3236263
22. Valentin, G., Zuliani, M., Zilio, D.C., Lohman, G., Skelley, A.: Db2 advisor: An optimizer smart enough to recommend its own indexes. In: Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073). pp. 101–110. IEEE (2000)
23. Watkins, C.J., Dayan, P.: Q-learning. Machine learning **8**(3-4), 279–292 (1992)
24. Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., Alonso, G.: Understanding replication in databases and distributed systems. In: Proceedings 20th IEEE International Conference on Distributed Computing Systems. pp. 464–474. IEEE (2000)
25. Zhang, J., Liu, Y., Zhou, K., Li, G., Xiao, Z., Cheng, B., Xing, J., Wang, Y., Cheng, T., Liu, L., et al.: An end-to-end automatic cloud database tuning system using deep reinforcement learning. In: Proceedings of the 2019 International Conference on Management of Data. pp. 415–432 (2019)