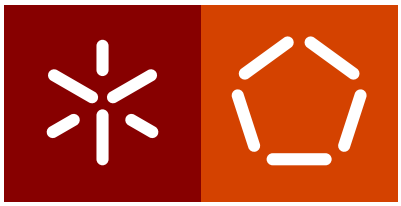
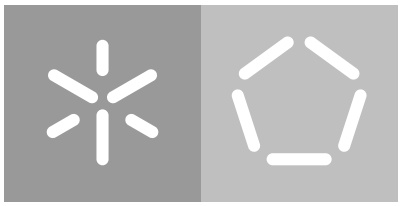


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luis Manuel Meruje Ferreira

Automatic Parameter Tuning Using Reinforcement Learning

December 2020



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luis Manuel Meruje Ferreira

Automatic Parameter Tuning Using Reinforcement Learning

Master dissertation
Integrated Master in Informatics Engineering

Dissertation supervised by
Doutor Fábio André Castanheira Luís Coelho
Professor Doutor José Orlando Roque Nascimento Pereira

December 2020

Despacho RT - 31 /2019 - Anexo 3

Declaração a incluir na Tese de Doutoramento (ou equivalente) ou no trabalho de Mestrado

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

AGRADECIMENTOS

Agradeço a todos os que tenham contribuído para a realização desta dissertação.

Primeiramente ao meu orientador, Doutor Fábio André Coelho, pelo conhecimento e disponibilidade prestados ao longo da elaboração da dissertação, e não só.

Ao meu co-orientador, Professor José Orlando Pereira, por pôr ao dispor a sua experiência e os seus conhecimentos que certamente fizeram com que esta dissertação seja melhor do que seria de outra forma.

À minha família pelo apoio e incentivo para fazer sempre as coisas com cabeça e dedicação.

A todos os meus colegas do HASLab, pelo bom ambiente de trabalho e por se mostrarem sempre disponíveis a ajudar.

Agradeço ainda ao INESCTEC pelo financiamento deste trabalho através da sua orçamentação plurianual sob o projeto INESTEC-UID/EEA/50014.

"The project AIDA - Adaptive, Intelligent and Distributed Assurance Platform (reference POCI-01-0247-FEDER-045907) leading to this work is co-financed by the ERDF - European Regional Development Fund through the Operacional Program for Competitiveness and Internationalisation - COMPETE 2020 and by the Portuguese Foundation for Science and Technology - FCT under CMU Portugal."

ABSTRACT

Every major *Database Management System (DBMS)* and most components in a distributed system in use today, closed or open source, comprise a set of configuration parameters which have substantial influence over the performance of the system. The correct configuration and tuning of these parameters often leads to a performance level that is orders of magnitude greater than that achieved by default configurations.

The number of parameters tends to increase as new versions are released. Moreover, the optimal values for these parameters vary with the environment, namely the workload to which the system is being subjected to, and the physical characteristics of the hardware it is running on.

It is common to delegate the responsibility of parameter tuning to a system administrator. The problem with this approach is that it requires both extensive prior experience with the specific system and workload at hand, and a large amount of the administrator's time. Moreover, variables may establish extensive and non-trivial correlations between them that are very difficult to identify and tune.

This dissertation introduces an automated and dynamic approach to parameter tuning using a reinforcement learning approach, while also adopting the use of deep neural networks to tackle the fact that complex relations between variables may exist.

Two use cases were implemented to showcase our approach, in the context of a distributed database. One where we adjust tuning variables specific to each replica and another where we adjust the shard configuration of the cluster (i.e. what shard is allocated to what replica). The reinforcement learning agents act at the middleware level, where all replication logic is held. The performance was measured in terms of the reward achieved by those agents as well as the values for the individual performance metrics that make up that reward. For the use case that concerns individual replica configurations, a maximum gain in reward of 105.41% was observed in one of the replicas as well as a maximum gain of 484.31% in one of the individual performance metrics. In the second scenario, of shard reallocation, we saw improvements in reward value up to 28.72% and of up to 69.92% for individual metrics.

Keywords – Reinforcement Learning, Distributed Databases, Middleware, Optimization, Machine Learning

RESUMO

Todos os principais sistemas de gestão de bases de dados, bem como a maioria dos componentes que são parte constituinte de um sistema distribuído em uso atualmente, licenciados ou abertos, incluem um conjunto de parâmetros de configuração que demonstram ter uma influência substancial sobre o desempenho do sistema. A correta configuração e ajuste destes parâmetros leva frequentemente a níveis de desempenho que podem ser ordens de magnitude acima do que os que são atingidos por configurações pré-definidas. O número de parâmetros tem tendência a aumentar à medida que novas versões são lançadas. Para além disso, os valores ótimos para estas variáveis tendem a variar com o contexto de execução, nomeadamente a carga de trabalho a que o sistema está a ser sujeito, e as características do hardware em que está a ser executado. É comum delegar a tarefa de ajuste dos parâmetros de configuração ao administrador do sistema. O problema com esta abordagem é que esta tarefa requer, por um lado, uma vasta experiência com o workload e sistema a ser configurado, e por outro, uma porção considerável do tempo do administrador. Para além disso, as variáveis podem estabelecer correlações complexas entre si que podem ser muito difíceis de identificar e compreender.

Esta dissertação apresenta uma abordagem automatizada e dinâmica para o ajuste de variáveis de configuração, recorrendo para isso a técnicas de aprendizagem por reforço e combinando estas com o uso de redes neuronais para abordar o problema de identificação de correlações entre variáveis.

Dois casos de estudo foram implementados para demonstrar a abordagem no contexto de uma base de dados distribuída. Um, em que são ajustados parâmetros de configuração individuais a cada réplica e outro onde se ajusta a configuração de shards do cluster (i.e. a que réplica está alocado cada shard). Os agentes de aprendizagem por reforço atuam ao nível de um middleware, onde é tratada toda a lógica de replicação. O desempenho foi medido em termos da recompensa alcançada pelos agentes assim como pelos valores das métricas individuais de desempenho que compõem essa recompensa. Para o caso de estudo relativo às configurações individuais de cada réplica foi observado um ganho máximo de 105.41% no valor da recompensa e um ganho máximo de 484.31% no valor de uma das métricas individuais. No caso de estudo de realocação de shards, foram observados ganhos no valor de recompensa de até 28.72% e 69.92% para métricas individuais.

Palavras chave – Aprendizagem por Reforço, Bases de Dados Distribuídas, Middleware, Otimização, Aprendizagem Máquina

Despacho RT - 31 /2019 - Anexo 4

Declaração a incluir na Tese de Doutoramento (ou equivalente) ou no trabalho de Mestrado

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

D

DBA Database Administrator.

DBMS Database Management System.

DDPG Deep Deterministic Policy Gradient.

DLSN DistributedLog Sequence Number.

DNN Deep Neural Network.

DQL Deep Q-Learning.

DRL Deep Reinforcement Learning.

G

GP Gaussian Process.

GRS Gaussian process Representation of a response Surface.

J

JDBC Java Database Connectivity.

L

LHS Latin Hypercube Sampling.

M

ML Machine Learning.

N

NP-HARD Non-deterministic polynomial-time hardness.

P

PB Primary-Backup.

R

RL Reinforcement Learning.

W

WAL Write Ahead Log.

1

INTRODUCTION

Configuration parameters are present in almost every software system that is currently used. As new software versions are released, the number of adjustable parameters in the system tends to increase, which makes the task of adjusting and tuning parameters increasingly harder. This is particularly true for database systems, where several distinct components for data storage and query execution offer a very diverse array of tunable configuration. As an example, consider the database engines PostgreSQL[20] and MySQL[18].

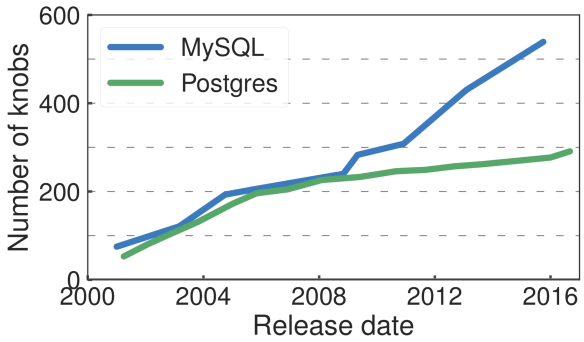


Figure 1: MySQL and Postgres number of configurations over time.

Figure 1, depicted in ottertune's article[27] (described in detail on section 2.3.2) shows the evolution over the number of configurations for MySQL and Postgres. From the early 2000's until 2016 the number of adjustable configurations for each one of these DBMSs has increased considerably, from less than 100 to over 500 in the case of MySQL, and from less than 100 to approximately 300 for the case of Postgres.

System configurations can be classified in two categories, namely: performance tuning parameters and setup parameters. Performance tuning parameters describe settings that have an impact over the performance of the system. Depending on the specific software being analysed, there can be hundreds, if not thousands of distinct variables of this type. They may be categorical, discrete or continuous in nature. Some common examples are thread pools sizes, buffers sizes or the maximum number of client connections allowed. Special care must be taken when setting these parameters, as certain values may cause the system to crash. For example, setting memory allocation limits to a value bigger than what the physical hardware can provide may cause such a failure. The setup parameters describe technical details required for a program to function properly. These include setting configuration file locations, interface addresses, logging level, among others. Generally speaking, these are deployment specific and are needed to make sure the components are set up

properly. They don't usually have an impact on performance, although there are some exceptions like the logging level. Some other examples include: from what interface and port a component should be available, how can remote services be accessed and what credentials should be used to access them or even where should data backups be stored.

The focus of this dissertation is on the variables that affect performance. As such, throughout the rest of this dissertation, any reference to a configuration parameter or variable should be understood as belonging to the first class described above.

The concept of optimizing parameter values is particularly interesting in the case of a distributed setup.

Consider a scenario where we have multiple database shards. In the context of a database, a shard represents an aggregate of data that can be separated from the rest of the data according to some criteria. For example, if a database models a chain of warehouses, data may be split by warehouse, or even by group of warehouses. Warehouses from the north region could be assigned to shard number 1, while shards from the south to shard number 2, etc.

This allows for horizontal scaling of the database. Horizontal scaling represents an increase in the database's storage size and often allows for higher processing power by using multiple machines (e.g. one shard per machine).

Each shard will have it's own set of data and will probably be subjected to a unique workload. They may even be hosted on machines with distinct hardware capabilities. Therefore, each shard will have different optimal parameter settings, which makes the task of parameter tuning harder. If we expand our scope to include the possibility of distributed transactions, finding those optimal settings may be even more difficult. Distributed transactions manipulate data from multiple shards at the same time, raising the level of complexity. A distributed transaction may cause one of the shards to wait for another to finish an operation. This means that the performance of a shard, and thus its tuning variables, will not only affect its own performance but potentially the performance of every shard involved in the transaction. Moreover, for certain types of cluster systems, there may be parameters with different scopes, as some may affect the entire cluster while others may affect only a single node or even a single user session. This is also an aspect that can be explored to further optimize the system at hand.

Currently, machine learning techniques such as reinforcement learning or deep learning are becoming very accessible and affordable, which motivates the increasing universe of domains where they are currently considered. Deep neural networks, while difficult to interpret, are good at finding complex patterns. That is why they are a good candidate for parameter tuning. They are better at identifying and modelling the interactions that are formed between different variables, than models that only identify low degree polynomial interactions.

Considering the fact that real database workloads tend to change according to, for example, the hour of the day or the time of the year (among others), we see that there is an additional factor to take into account. That is, that workloads may vary with time. To tackle this side of the problem, *Reinforcement Learning (RL)* presents as a good candidate. Briefly, RL starts from a base policy, or decision making strategy, which it adjusts according to feedback that it receives from the target environment (the database in this case). The policy can correspond to something simple such as a table (Q-learning), or to something much more complex such as a *Deep Neural Network (DNN)*. So, when the workload

to which the database is being subjected changes, the feedback for the RL agent will also change and, consequently, so will the agent's policy.

Currently, configuration parameters are seen as static elements of a system. The state of the art analysed in section 2.3 often considers a period of time to find optimal parameters, and then simply applies the best configuration found to the database. By considering Reinforcement Learning techniques the process can be taken a step further, by continuously adapting the system configuration to changes in the workload. This means that instead of having a static parameter setup, a dynamic one can be established. Combining Deep Neural Networks and Reinforcement Learning, called *Deep Reinforcement Learning (DRL)*, provides a very interesting combination of techniques for database parameter tuning.

1.1 PROBLEM

This dissertation addresses the problem of automatic and online parameter tuning. The responsibility of setting the values for configuration parameters is often given to the system administrator. Many times, administrators don't have enough know-how to set the correct parameters, or the process becomes time-consuming. In such cases, a common practice is to use default values, which has shown to be sub-optimal in most cases [30][27][10][31][16]. Setting adequate values for configuration parameters is difficult given the sheer amount of parameters which can go up to hundreds and the complex nature of the task, in which the values set for one parameter may impact the values of another. Moreover, adjusting configuration parameters often leads to the need to restart the underlying system, which for many applications that require, for instance, high availability (e.g., cloud databases), may cause unwanted downtime.

1.2 OBJECTIVES AND CONTRIBUTIONS

This dissertation introduces TuneRL, an automatic parameter tuning system that is tailored for a database replication middleware. It explores configurations at the shard and replica levels by employing Deep Reinforcement Learning. Besides the implementation of two use cases for automatic tuning, contributions are also made at the level of the middleware itself.

First, two new features were added to the middleware, namely a monitoring system that allows for the collection of state metrics and an update system that allows to remotely update the configuration parameters without stopping or restarting any part of the underlying system.

Second, two use cases are introduced to support the system description, implementation and evaluation. In the first use case, a replica RL agent is built to configure replica specific variables. In the second use case, a shard management RL agent is built, determining, at each moment, how many shards each replica should control. In order to allow the agent to migrate shards between replicas, a shard reallocation mechanism is implemented within the middleware. The reallocation of a shard does not incur in a restart of any of the system's components during its operation.

With the use of agents that continuously adapt configuration variables to the workload we introduce the concept of dynamic parameter tuning, as opposed to current static configurations.

1.3 DOCUMENT OUTLINE

This document is organized as follows. Chapter 2 introduces and contextualizes current State-of-the-art and Related Work. Chapter 3 introduces the target middleware being tuned and delineates contributions made to this middleware during the process of implementing the proposed use cases. Chapter 4 introduces TuneRL while Chapter 5 evaluates it. Finally, Chapter 6 concludes this dissertation and drafts new research opportunities as future work.

2

BACKGROUND

This chapter introduces and discusses two main topics that are required to support the the contributions presented. It addresses automatic configuration tuning concepts and candidate machine learning solutions. Finally it merges and discusses these topics and presents meaningful related work.

2.1 AUTOMATIC CONFIGURATION TUNING

There are already a number of automatic configuration tuning solutions for various systems [30][27][10][31][16][8]. These solutions can be divided into two types: search-based, such as bestconfig [31] or SARD [8] (Section 2.1.1) and model-based, namely iTuned [10] (Section 2.3.1), Ottertune [27] (Section 2.3.2), CDBTune [30] (Section 2.3.3) and QTune[16] (Section 2.3.4).

Configurations are chosen by comparing samples. In the context of database parameter tuning, a sample is obtained as follows:

- A set of configurations is chosen and applied to the database instance we want to tune.
- A workload is run on that instance.
- Performance metrics relative to the execution of the workload are registered.

The sample is composed by the ensemble of variable values, workload and performance metrics.

The process of obtaining a sample is sometimes referred to as an experiment. The logic followed to choose the set of values to use in the next experiment is called the sampling strategy. For samples to be comparable, workloads have to be identical between them. When samples are compared, the one with the best performance metrics (e.g. lower latency, higher throughput) is considered to be better or of higher quality than the other one. The objective is to obtain the most high quality samples with as few experiments as possible. At the end of the tuning process, the set of values from the highest quality sample obtained is applied to the production database. This is a trial and error process, where the chosen set of values must be within the acquired samples.

2.1.1 *Search-based solutions*

Search-based solutions differ in their sampling strategy. Samples can both be chosen randomly or the search space be divided into sub-spaces and random sample search be conducted from each sub-space. Usually, this type of solutions are not feedback driven. In other words, the choice of the next experiment to run is not affected by the results gathered from previous experiments. However,

they may have memory, such as in the case of Latin Hypercube. Two examples of sampling strategies were examined, Latin Hypercube and SARD [8].

Latin Hypercube

Latin Hypercube is used to generate random samples from a multi dimensional search space. The idea is to distribute samples more evenly, when fewer samples exist. This method is referred to as a good solution for Monte Carlo simulation, since it reduces processing time by up to 50 percent [24].

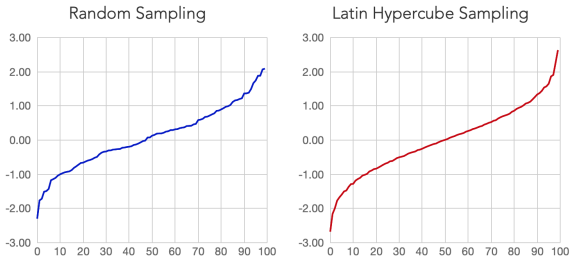


Figure 2: Random Sampling vs Latin Hypercube Sampling(LHS), 100 Samples.

Figure 2 was generated on [22], depicting that with only 100 samples, the distribution is much smoother when *Latin Hypercube Sampling (LHS)* is used in comparison to when Random Sampling is considered. As the number of samples increases, the lines will become similar, but LHS's line will always be smoother. In practice, samples will be more evenly spaced instead of having "hot spots", intervals of values where there are more samples than average.

Let us look at an example for two variables from Tomcat [12], a Java HTTP Web Server. We can visualize the sampling process as Table 1 (2D representation) with a set of rows and columns. Each axis represents a variable to be sampled. The horizontal axis will be *maxConnections* and the vertical axis *socketBuffer_Byte*. Each variable is split into M equally likely intervals of values. The number of samples will be equal to M. The rule is that there can only be one sample per column and row. Within each cell selected, a random sample is taken from each variable's interval.

Table 1: **L**atin Hypercube Sampling methodology.

socketBuffer_Byte	0-7.5K	7.5-15K	15-22.5K	22.5 - 30K
75-100K	X			
50-75K			X	
25-50K		X		
0-25K				X

maxConnections

Each **X** marks a sample. If one more variable was considered, the sampling process would be represented by a cube (3D). For a higher number of variables, there is no simple visual representation, as we would be dealing with hyperplanes.

SARD

SARD [8] is a method designed to rank database tuning parameters. It runs a series of experiments and evaluates the importance of each variable based on it's impact on performance. For each variable it considers only the minimum and maximum values, as it assumes these are the values that will provoke the biggest response from the system. One of the advantages of SARD is that it runs a linear number of experiments, even if variables are continuous in nature. A very important assumption that should be noted is that "the provoked response, such as the total execution time, is a monotonic function of the input parameter values". So, SARD assumes that when we increase the value of a variable, the provoked response will also, and always, increase/stay the same or always decrease/stay the same, which is not always the case. The response surface of a tuning variable in relation to the system's performance may be non-linear.

SARD runs X experiments, where $X = (floor(N/4) + 1) * 4$, being N the number of variables to rank. The maximum and minimum values of a variable are described as $+1$ and -1 respectively. To determine which experiments to use, SARD uses a somewhat complex process. It builds a matrix, where each row represents a set of values for the variables with which to experiment. The first row is obtained from the recommendations of the Placket & Burman designs [19]. These are comprised, in part, by a series of parameter value recommendations, where possible values for the variables are either $+1$ or -1, for X = 8,12,16,...,96,100. After obtaining the first row, the next $X - 2$ rows are obtained by shifting the values of the immediately previous row to the right, in a cyclic manner. The values of the X-th row are all set to -1. To improve the accuracy of variable rankings, the number of experiments may be doubled, by inverting the signs of each of the first X rows (designated as foldover).

An experiment is comprised of executing a certain workload with the parameter values set according to a row of the matrix. Experiment i will use row i. The execution time of each query in the workload is registered. For each variable, results are combined by multiplying it's value, $+1$ or -1, by each queries execution time, and then summing those products. The absolute value of the result is the metric taken into account when ranking.

As these is a rather complicated process, Figure 3 shows an example matrix from the original article [8]. Letters A through G are variables, and Q1, Q2 and Q3 are queries from the workload.

	A	B	C	D	E	F	G	Execution Time		
								Q1	Q2	Q3
R1	+1	+1	+1	-1	+1	-1	-1	34	110	10.2
R2	-1	+1	+1	+1	-1	+1	-1	19	72	10.1
R3	-1	-1	+1	+1	+1	-1	+1	111	89	10.3
R4	+1	-1	-1	+1	+1	+1	-1	37	41	10.3
R5	-1	+1	-1	-1	+1	+1	+1	61	96	10.2
R6	+1	-1	+1	-1	-1	+1	+1	29	57	10.2
R7	+1	+1	-1	+1	-1	-1	+1	79	131	10.3
R8	-1	-1	-1	-1	-1	-1	-1	19	47	10.1
R9	-1	-1	-1	+1	-1	+1	+1	135	107	10.3
R10	+1	-1	-1	-1	+1	-1	+1	56	74	10.3
R11	+1	+1	-1	-1	-1	+1	-1	112	48	10.1
R12	-1	+1	+1	-1	-1	-1	+1	74	91	10.1
R13	+1	-1	+1	+1	-1	-1	-1	55	99	10.3
R14	-1	+1	-1	+1	+1	-1	-1	117	123	10.1
R15	-1	-1	+1	-1	+1	+1	-1	51	77	10.3
R16	+1	+1	+1	+1	+1	+1	+1	76	81	10.2

Figure 3: Example SARD matrix.

If we multiply each variable's column with the queries' columns, sum the results for each variable, and then convert all signs to positive, we get the table in Figure 4 (also from the original article). We can conclude, for example, that variable D has the biggest impact on the execution time of Q1.

	A	B	C	D	E	F	G	stdev
Q1	109	79	167	193	21	25	177	136.4
Q2	61	161	9	143	39	185	109	123.3
Q3	0.40	0.80	0.00	0.40	0.40	0.00	0.40	0.44

Figure 4: SARD: computed score of variables' influence on query execution time.

The most obvious problem we can find with this method is in how to treat categorical variables. How do we determine which category represents -1 and which represents $+1$? Not only that, but due to the assumptions made about monotonicity, every category would have to be ordered.

2.1.2 *Model-based solutions*

From existing literature, model-based approaches outperform search-based ones. The reason for this is that, contrary to model-based methods, search-based methods do not try to explain or, more accurately speaking, do not specify how changes in a variable or variables affect overall performance. They simply follow a predetermined strategy to choose experiments, hoping that a new sample will be better than the ones obtained so far.

Within model-based approaches, Machine Learning methods, and in particular, Deep Reinforcement Learning methods have the best results overall [30][16]. Intra-variable relations and relations between variables' values and overall performance of the system can be non-linear. Machine learning methods

are known for being able to solve complex problems and identifying complex patterns, where more conventional methods prove to be insufficient. Some examples of this are, for example, random forest models, that utilize an ensemble of decision trees to find answers for regression, classification and other problems, or convolutional neural networks for image analysis. DNNs are especially good candidates, as they are known to be good at finding or representing non-linear patterns. RL is a class within machine learning methods that is based on actions and corresponding rewards. This means that instead of relying in large datasets of historical data to train our models, we can use feedback from the environment itself. Multiple RL methods, in turn, use Deep Neural Networks, e.g. Deep Q-Learning and *Deep Deterministic Policy Gradient (DDPG)*. Merging DNN and RL we get DRL. Here, we can benefit simultaneously from the ability to find non-linear relations and from receiving feedback directly from the environment we are trying to tune. RL decides which actions to take, runs it's own experiments, assesses what the consequences of it's actions are and corrects it's behaviour accordingly. This eliminates the need for a dataset to train the model.

Moreover, reinforcement learning (RL) methods allow us to continually adjust the DBMS tuning parameters so that when the workload changes, the configurations can also be changed accordingly, in real-time.

Some real examples of model-based parameter tuning systems are explored in detail in section 2.3

2.2 Machine Learning Solutions

Machine Learning (ML) is a subset of algorithms within Artificial Intelligence that improve when given more data. In a sense, they learn from experience, achieving better results as they get fed more data. ML is good at solving very complex tasks such as classifying images, analysing text, dividing data according to patterns, financial forecasting, weather forecasting, among others.

ML can be divided into three types: supervised, unsupervised and reinforcement learning.

Supervised learning is arguably the most iconic of the three. Usually, there is a dataset with a group of input parameters and corresponding output values. The objective is to train a model on that dataset, and then allow the model to predict the correct output value when presented with previously unseen input values. This type of machine learning is used to solve classification and regression problems. Some applications include image classification, market price prediction, spam filtering, among others.

Unsupervised learning follows a very different approach. Data is composed by a set of observations with no label or output. The objective is not to make predictions, but to find patterns or similarities in data points. There is no model training phase. An example of unsupervised learning is the *k-means clustering*, where n observations are divided by k clusters based on the distance between the data values of the observations.

2.2.1 *Reinforcement Learning*

Reinforcement learning is a different approach to ML. It uses reward based learning, instead of pre-acquired data to train the model. An agent observes the state of an environment, decides on what action to take and then receives a reward for his action that can be either positive or negative (as well as the environment's updated state). It then adjusts the action policy according to the rewards

received. The objective of reinforcement learning is to maximize rewards. Something that is special to some Reinforcement Learning algorithms is the idea of future-reward, as actions are adjusted not just for immediate reward but for long-term rewards, allowing the balance between them to be adjusted. The hyper parameter that controls how much future reward is favoured versus current reward is named gamma.

Figure 5 describes the basic idea of reinforcement learning.

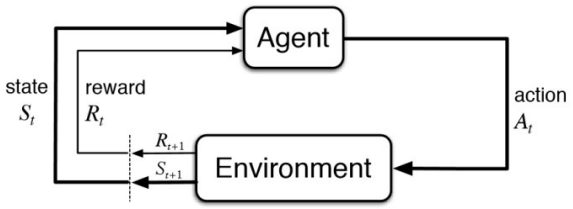


Figure 5: Reinforcement Learning representation.

Source: kdnuggets

The **Agent** monitors a certain **Environment**. For a given **State** the agent decides on an **Action**, by means of it's **Policy**. A Policy maps States to Actions and the objective of Reinforcement Learning is to find the best possible Policy. The best Policy is the one that maximizes a **Reward**, so policies are adjusted according to Reward values. Reward values are calculated after an Action is applied to the Environment, according to the feedback given by the Environment, using a **Reward Function**. Each cycle of retrieving the state, mapping that state to an action and then updating the agent's policy according to the observed action reward is called a **Step**. The period of time comprised by a series of steps that accompany the environment from an initial state to a final one (e.g., from the first move of a chess game to a checkmate) is called an **Episode**. When trained through enough episodes in different situations, an agent can be able to transfer what it learned from a context to another. For example, by building an agent to play chess and letting it train with different skilled opponents or even against himself he might be able to defeat other opponents including even the chess world champion.

2.2.2 *Q-Learning*

Let's take a look at one of the simplest and most well known Reinforcement Learning algorithms, Q-Learning. In Q-Learning a table defines the agent's policy. Columns describe the possible actions and lines the possible states. Table entries are called q-values.

Q-table initialised at zero

	UP	DOWN	LEFT	RIGHT
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0

After few episodes

	UP	DOWN	LEFT	RIGHT
0	0	0	0	0
1	0	0	0	0
2	0	2.25	2.25	0
3	0	0	5	0
4	0	0	0	0
5	0	0	0	0
6	0	5	0	0
7	0	0	2.25	0
8	0	0	0	0

Eventually

	UP	DOWN	LEFT	RIGHT
0	0	0	0.45	0
1	0	1.01	0	0
2	0	2.25	2.25	0
3	0	0	5	0
4	0	0	0	0
5	0	0	0	0
6	0	5	0	0
7	0	0	2.25	0
8	0	0	0	0

Figure 6: Example of a Q-table.

Source: towardsdatascience

Figure 6 depicts an example of what this q-table can look like. At the beginning of the training process, the table is empty and all q-values are equal to zero. The policy states that for each state we should choose the action with the biggest corresponding q-value, as this is the one that is expected to provide the biggest reward. Since at the beginning all values are 0, actions are chosen at random. Once actions are executed, the resulting reward values are used to update the q-values. As depicted, at the end of the training process the system learns that the best action, while in state 0, (what that state means is irrelevant) is to take a left. If the system always chooses the action with the highest q-value, it could get stuck with this action, and never discover that the true q-value of a different action that was never tried, is actually higher. For this reason, a percentage (e.g., 10%) of actions are chosen at random, not by q-value. This means that q-learning employs an *exploration vs exploitation* type of strategy.

In practical terms, Q-Learning has very serious limitations. It can only deal with a finite amount of states and actions. Also, if the number of actions and states is finite but very big, the table will be too big to handle. This means that, unless we sample databases' states and their tuning parameter values, Q-Learning is not a valid solution for the problem at hand. The process of sampling itself would be an additional source of complexity and debate.

2.2.3 *Deep Q-Learning*

The *Deep Q-Learning (DQL)* algorithm replaces the Q-Learning's table by a deep neural network. This algorithm presents as an improvement over the previous one, in the sense that it can handle continuous state spaces. Each input node represents a state variable that can be continuous in nature, while each output node represents a possible action and the value calculated by the network for that output node will be the q-value of the action. The action chosen will be the one with the highest q-value, just as in Q-learning. Rewards are used to adjust the weights of the neural network by back-propagation.

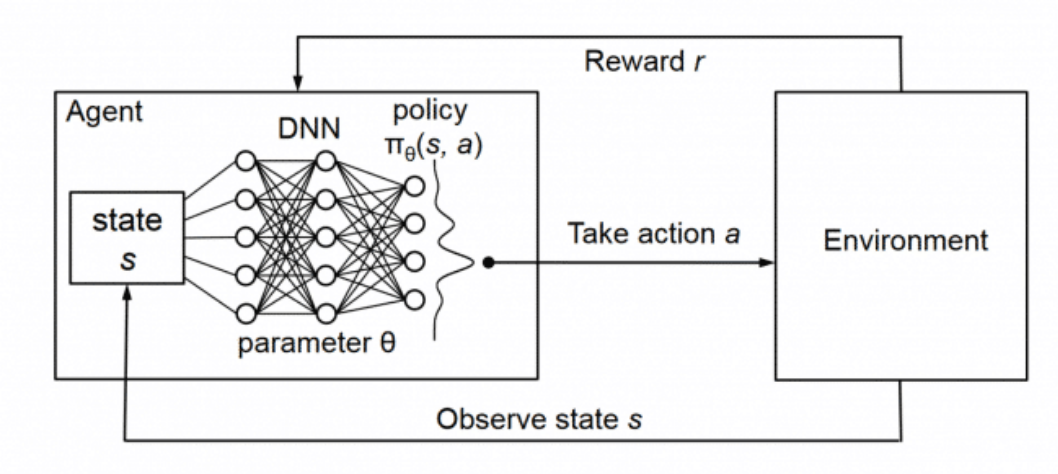


Figure 7: Deep Q-Learning diagram.

Source: novatec-gmbh

Figure 7 shows a summarized view of the inner workings of the agent in the context of DQL. The state is fed to the input nodes, where each node corresponds to a state variable, which is then mapped to it's corresponding q-values for each possible action, the values of the output nodes. The action with the highest q-value is applied to the Environment.

2.2.4 *Deep Deterministic Policy Gradient*

DDPG[17] can be seen has an extension of DQL to continuous action spaces. In Deep Q-learning, the neural network takes as input a state, and outputs the q-values of each possible action. The problem is that, when we have a continuous action space, we have an infinite number of possible actions. That would mean we would need to have infinite output nodes, in order to calculate the q-values of every possible action which is obviously impossible. For this reason, DDPG opts instead for an actor-critic architecture, where the actor and the critic are distinct neural networks. Actor-Critic algorithms aim to take advantage from the highlights of both value-based and policy-based while eliminating all their drawbacks. The actor is our policy, it still takes the current state as input, but instead of outputting q-values for actions, the output nodes give actual continuous values for actions to take. For example, the values of the output nodes may be values of tuning variables. The critic takes as input the state and the actions given by the actor. It's output is a q-value that essentially says how good the actions chosen by the actor are. It then calculates the gradient of the q-value in respect to the given action, which is then used to calculate an error that will, in turn, be used to improve the actor network. In regard to training the critic, in simplistic terms, the reward value, together with other factors, are used to improve the critic's network.

2.3 RELATED WORK

2.3.1 *iTuned*

iTuned is a tool, presented in 2009 [10], that aims to automate database parameter tuning. It runs multiple experiments, in order to build response surfaces that represent how changing a variable will affect performance. Experiments are chosen based on an expected improvement metric, which tries to explore areas of the surface with high uncertainty. This tool's most notable feature is its use of idle resources to run the experiments. Using this strategy allows to both improve system performance and reduce resource under-utilization, without introducing any significant overhead. Bellow are iTuned's three major components:

- Planner
  Through a technique they call *Adaptive Sampling*, the planner chooses the next experiment to run. The idea is to find the experiment which will be the most useful to better determine the shape of a response surface. It makes use of the information gathered by past experiments. Experiment planning is preceded by an initialization process based on *Latin Hypercube* sampling(2.1.1). The authors consider *Adaptive Sampling* to be an improvement over both Grid sampling and SARD[8] because, contrary to both this techniques, theirs is feedback-driven.

- Executor
  The executor hunts for idle time, in order to run experiments during those periods. It can run experiments on production, hot standby, testing or even staging databases based on permissions given by the user. The overhead incurred by this component on the performance of the production workload is said to be low. The executor may run multiple experiments at the same time, if multiple databases are available. Figure 8 depicts the representation of the executors architecture.

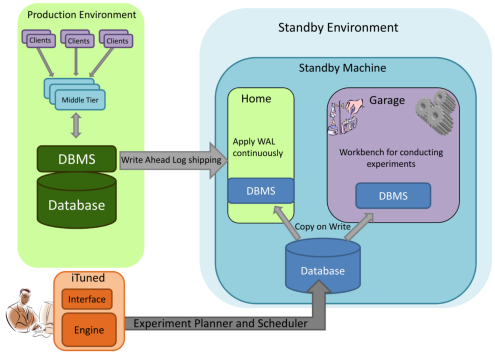


Figure 8: iTuned's architecture.

It uses a mixture of database snapshots, copy-on-write and the Zettabyte File System [5] to ensure that experiments have a low overhead and do not corrupt the data of the actual database.

- *Gaussian process Representation of a response Surface (GRS)*
  Gaussian process Representation of a response Surface is used to allow the visualization of response surfaces, from the experiments run so far, with confidence intervals on estimated

performance. It also allows to distinguish which variables have the biggest impact on overall performance, see how variables interact with each other and recommend settings. Finally, GRS also aids Adaptive Sampling in determining the expected improvement of a certain experiment.

While iTuned has been surpassed in terms of performance gain [27] it introduced the idea of using idle resources for exploring new configurations, thus reducing both resource under-utilization and the risk associated with experimenting with new configurations in production databases. This strategy was not implemented in our use cases, but we leave to future work the use of secondary database instances for model training, mainly for the reason of decreased risk in production environments.

2.3.2 *Ottertune*

Ottertune was presented in 2017 [27] and represents a substantial improvement over previous tools. It's idea is to form a pipeline composed of various machine learning methods to achieve automatic parameter tuning on database management systems. The original paper's evaluation section shows that the system can find better configurations and in a shorter period of time than iTuned (2.3.1) and other alternatives. In order to achieve this, it requires a large and good quality dataset, that has diverse samples and is representative of the system at hand. The dataset is comprised of configuration variables and corresponding performance pairs. The system works with multiple database engines.

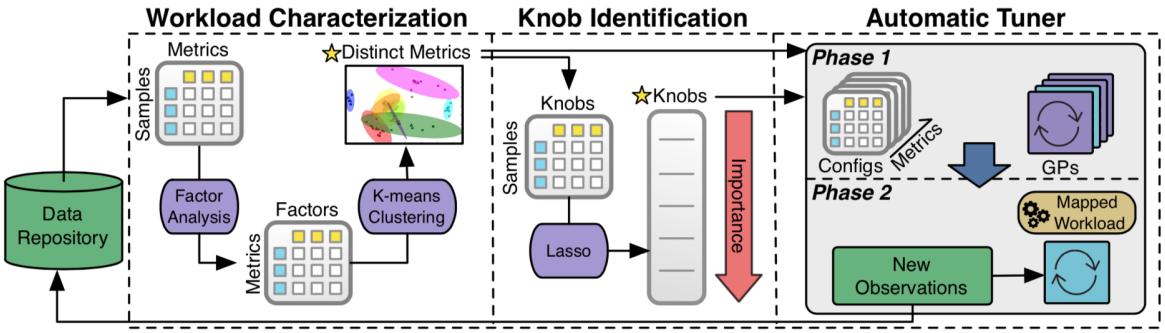


Figure 9: Ottertune's architecture (from the original paper).

Figure 9 depicts the system architecture. The first step in ottertune's pipeline is to identify the most relevant metrics for the DBMS at hand, in order to reduce their dimensionality. This helps to determine a small group of metrics that still represent all the relevant aspects of the systems functioning, contributing to reduce training and predicting times during later phases. In order to achieve this, they first apply *Factor Analysis* to group strongly correlated metrics into a single factor (which is a linear combination of the metrics it represents). After that, k-means clustering is used to group similar configuration variables, and a single metric is chosen from each cluster (the one closest to the middle of the cluster).

The second step is then to select and rank configuration variables (or *knobs* as they call it), which will also reduce training and predicting times later on. The ranking of variables is done using LASSO [25], a variation of linear regression where the model is penalized by the sum of the absolute values of the weights for the independent variables. Some polynomial factors, which represent correlations between variables, may also be added to the LASSO process in order to capture

interactions between those variables. If they get ranked high, it means there is a correlation between them, and that correlation is significant for the system's performance. During a tuning session, knobs are incrementally added, until there is no significant improvement in performance from adding them. While the idea of capturing correlations between variables is both interesting and relevant in the case at hand, polynomial factors may not be enough to fully identify the complex interactions established between them. For this particular aspect, the use of deep neural networks may be a better alternative.

After selecting the most relevant metrics and knobs, Ottertune tries to match the current workload of the system to the most similar one it has seen previously. This match may change during the initial tuning phase, but stabilizes as more data is gathered. Then, it resorts to a technique called *Gaussian Process (GP)* to effectively choose a configuration. The selection of configuration is based on previous results from the workload it matched to. GP follows an *exploration vs exploitation* strategy, using gradient descent to explore local optimums, for the case of exploitation. As more tuning configurations are introduced, they and their corresponding performance metrics are stored in order to further refine the suggestions given.

The concept of identifying the most important metrics and configuration variables is very important in the context of machine learning. By using a smaller number of input and output variables, we can find simpler models that take less time to train and still make accurate predictions. Even so, being able to include more tuning variables in our models and to have the system understand the complex relations between them may lead to even better results. In the case of Ottertune, data is divided by pre-defined hardware profiles, major DBMS version and workload. This means that, although it is able to tune different DBMSs, a very large dataset is required, with distinct entries for different combinations of those factors. If we consider a cloud environment, this is especially problematic due to the large possible combinations of hardware features and DBMSs.

Ottertune was the first notable application of machine learning to automatic parameter tuning. However, it is very complex to understand and hard to implement. For this reason, the original idea of this dissertation was to propose an alternative to it, based on DRL, but that plan was later scratched due to the publication of CDBTune [30].

2.3.3 *CDBTune*

CDBTune was introduced in 2019 [30] and was the first to apply Reinforcement Learning to the problem of automatic parameter tuning in the context of DBMSs.

OtterTune made use of multiple machine learning methods in order to tune a DBMS. Even though it is an impressive use of artificial intelligence concepts, it's not without it's flaws. First, being able to conjugate the use of all the methods is complex. Moreover, while techniques such as LASSO [25] allow to reduce the number of variables used and also find correlations between variables, which in turn may make it easier to find good configurations, they may also cause the system to overlook even better ones. Correlations between variables can be non-linear and not representable by the polynomial expressions that were used.

CDBTune opts for Reinforcement Learning with the DDPG approach. DDPG is an actor-critic model which uses two deep neural networks to effectively recommend actions. This gives it some advantages over Ottertune's system. Firstly, deep neural networks are known to be good at modelling complex patterns of interaction between its input variables, versus the Lasso method in Ottertune

where correlations have to be explicitly introduced. Not only that, but the ranking of configuration variables is made implicitly. Furthermore, it is able to take into account all of the metrics given by the database while also attributing them different levels of importance. This avoids the use of techniques to reduce the number of state metrics, such as Factor Analysis and k-means clustering in the case of Ottertune. CDBTune also has a better adaptability. The same model can be used in different hardware environments and still achieve good results in both cases.

While Ottertune's evaluation was conducted by using several DBMSs, CDBTune only considered the CDB DBMS. But, despite its lack of results with different DBMSs, CDBTune claims to achieve better results than Ottertune, which is also supported by more recent articles [16]. CDBTune's DDPG model uses the internal database metrics as it's state. The reward is calculated using external metrics, such as latency and throughput. The reward tries to mimic the reasoning of a *Database Administrator (DBA)* when tuning a database. It calculates the improvement in performance between the current step and the moment before the tuning session was initiated. It also calculates the improvement between the current step and the previous step. The weights of each of these values in the final reward value can then be adjusted.

CDBTune is the solution that most resembles the use cases proposed in this dissertation. Due to the publication of this paper during the state of the art research phase, a path towards distributed databases was taken instead of considering single instance databases.

2.3.4 *QTune*

Introduced shortly after CDBTune, QTune [16] uses very similar ideas, but with a very important difference over the former. It extracts features from queries into vectors in order to distinguish them and to identify key elements in the query that may hint at what configuration better suits it. In order to do so, it uses information from the database's engine query optimizer. This system also uses Reinforcement Learning for deciding on what the best configuration is in a given situation. The algorithm used is DS-DDPG, a variation of DDPG that is proposed in the article.

QTune allows for three different levels of tuning. Query-level tuning finds a good configuration for each query, which allows for low latency, but also low throughput since this mode doesn't allow for parallel processing of queries. Workload-level tuning allows for high throughput but also high latency. This method tries to find the best compromise in terms of configuration when all queries have to share the same parameter values, but since the configuration isn't optimized for any particular query, latency may not be optimal. Finally, cluster-level tuning is a comprise between the previous two, where queries are divided into clusters and then each cluster will have it's own configuration. Queries from different groups can be run in parallel, which is said to result in both high throughput and low latency.

While the use of query information gives QTune an advantage over CDBTune, acquiring data from the engine's query optimizer comes at the cost of requiring good knowledge of the engine's internal working. This means that QTune will not integrate easily with different DBMSs.

Due to the inherent attachment to the inner workings of the underlying database instance, use of query optimizers information for further improvement of the implemented use cases (4) was discarded, since the objective was to find solutions where the environment for the RL agents was

restricted to the middleware shown in 3.2. This allows for a higher interchangeability between DBMSs.

3

REPLICATION MIDDLEWARE

Here is presented the middleware that is being optimized in the proposed use-cases. Additionally, relevant background information and contributions made to the middleware are also described.

3.1 DISTRIBUTED DATABASES

A distributed database must have more than one node or database instance. We refer to these nodes as **replicas**. Usually, each replica will be deployed in different hardware in order to leverage the computing and storage capacity of multiple machines. This is called horizontal scaling, as opposed to vertical scaling where the hardware of the machine where the database is held is upgraded. Vertical scaling is limited by the best available hardware, while horizontal scaling is only limited by the amount of machines we can coordinate. For horizontal scaling, the higher total computing capacity allows for higher throughput, since reads and sometimes writes can be conducted in different replicas at the same time, depending on consistency constraints. Distributed databases may also promote lower latency, since different replicas can be located in different geographical locations allowing them to be closer to the users. In order for a distributed database to work there must be a communication protocol in place that caters to the specific needs of the system at hand. This communication primitives allow for data exchanges to be conducted with certain guarantees, essentially assuring data consistency between the replicas. When discussing distributed database configurations, a series of topics are often considered, namely Replication and Sharding, which are briefly introduced.

3.1.1 *Replication*

Data replication refers to having more than one copy of the same data in different replicas, in order to provide high-availability. If one of the replicas happens to crash or become unavailable for some reason, the data can be serviced from one of the remaining available copies. There are two main types of replication: active and passive replication.

Active Replication

In active replication, each client request is executed by all the replicas. This type of replication requires operations to be deterministic. This means that given the same initial state and request, executing that request will yield the same final state in all replicas (e.g., random operators are not allowed). Moreover, requests must be received in the same order by all replicas, in order to guarantee

consistency. To transmit a client request in the same order to all replicas, an Atomic Broadcast communication primitive is used [3][7]. This primitive has two properties: Either all replicas receive a message, or none does; Messages received by replicas are received in the same order. In active replication, any replica that has finished executing a request may reply to the client that sent that request.

Passive Replication

Passive replication does not require deterministic operations. Client requests are processed on a single replica, the primary. Only the resulting state changes are replicated and applied to the remaining replicas (backups). Only after all backups have confirmed that they have updated their state can the primary, and only the primary, reply to the client. To transmit state updates to the backup instances, the View Synchronous Multicast communication primitive is used, which is a group membership service. View Synchronous Multicast states that if process P is valid in view V and sends a message M to V, then all members of V either receive M or install a new view V'. Contrary to what happens in active replication, this communication primitive is not available to the client, it is implemented on the server's side.

When comparing these two methods, aside from the requests' determinism, active replication does not imply an interruption of service if one of the replicas fails. Replica failure is transparent to the client, as long as at least one replica remains active. On passive replication however, if the primary fails, there will be a period where a new primary is being elected, and thus processing of client requests will be interrupted during that time. If one of the non-primary replicas crashes, the cluster can still serve client requests, without any major interruption. Other problems may arise with the use of passive replication, such as increased latency and need for transaction reissuing. If the primary fails while a transaction is being processed, the client must learn the identity of the new primary and reissue the transaction.

There are some alternatives that combine the use of active and passive replication, such as is the case of the experimental database replication middleware considered in this dissertation [11] (3.2).

When looking specifically at DBMS replication protocols, often one of the above mentioned methods is used as a reference point, but there are a multitude of variants or hybrid solutions. As such, there are a multitude of replication algorithms that differ between them according to consistency, performance, failover and recovery needs.

In the context of *Primary-Backup (PB)* architectures (passive replication), updates may be done synchronously (favouring consistency) or asynchronously (favouring performance). These are adequate for non-deterministic workloads and may be extensible to multi-master configurations when using data partitioning or reconciliation techniques (allowing for replicas' states to diverge).

The term commonly used for DBMSs that adopt the active paradigm is replicated state-machine. As stated above, this approach gives us client-side transparency in terms of failures, increased simplicity, and when compared to synchronous PB, lower latency. However, it requires processing to be deterministic.

3.1.2 *Sharding*

Sharding is the concept of partitioning big chunks of data into smaller, more manageable pieces, that can be distributed among multiple database instances. For example, considering 3 database instances that store information about 10 warehouses, it is possible to split data in a per warehouse strategy (assuming that warehouses' data is independent) and assign data from warehouses 1 through 4 to instance 1, 5 through 7 to instance 2 and 8 through 10 to instance 3. One of the advantages of sharding is that it becomes possible to have multiple database instances answering write requests at the same time. Data partitioning in a PB setup, described in the previous section, is one such example, providing us with what is called a multi-master system. Besides, the main reason for the use of sharding in a database context is for horizontal scaling. Each instance of the previous example may be deployed in a separate host, increasing the amount of processing, memory and storage capacity, without having to invest in powerful hardware that is often prohibitively expensive or just simply not available (vertical scaling). A side effect of sharding is the emergence of distributed transactions, since a transaction may require access to warehouses that are hosted on different instances.

Systems like Dgraph [9] or the middleware on which the use-cases are based (3.2), offer both sharding and replication at the same time, for example.

3.2 ARCHITECTURE

The automatic parameter tuning solutions presented in this dissertation considers a replication middleware as the base system. The middleware leverages collaboration work held as part of CloudDBAppliance project [1], particularly in [2]. It consists of a hybrid database replication system, composed of a distributed log deployment [4] and a group of replicas.

The replication middleware system considered is depicted in Figure 10. It is split into two parts: the client, and the middleware that interacts directly with the database's default *Java Database Connectivity (JDBC)*. The middleware entails two assumptions without which it may not be able to perform it's functions. Data should be partitionable, so that distinct data classes can be assigned to different replicas and transactions from different classes be conducted concurrently. Although the middleware can handle non-partitionable data, it's operation would be restricted to passive replication, which denies the performance benefits that the hybrid architecture of the middleware provides. Also, a write-set retrieval service in the underlying database service is required so that updates may be shared among the replicas, for replication purposes.

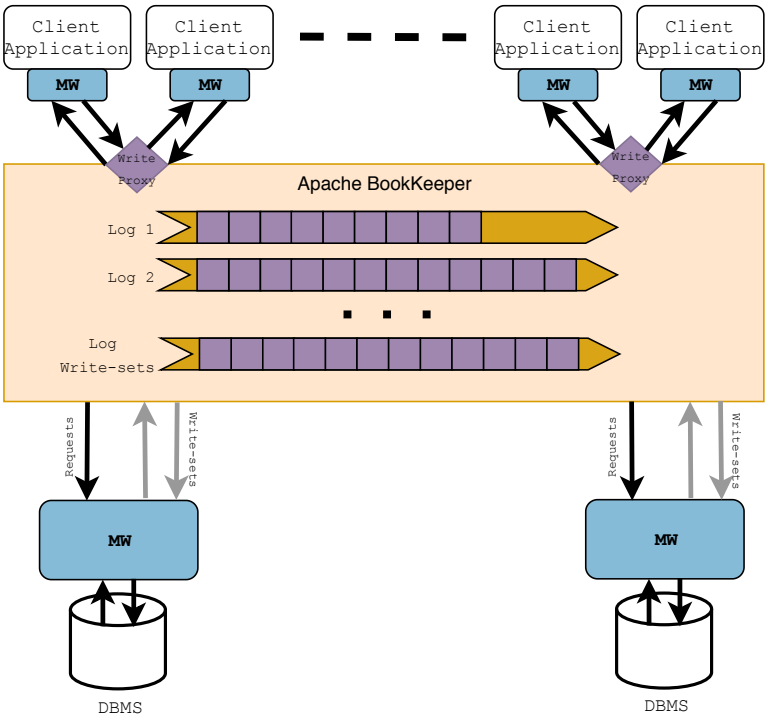


Figure 10: Comprehensive view of the replication middleware.

The system portion that attaches to the client is provided as an interface that allows the seamless execution of generic queries. The middleware delegates the persistency to the write-proxies provided by the Apache DistributedLog. Moreover, the write-proxies guarantee a distributed write into durable storage, and the middleware is notified ensuring the query execution in the underlying relational database, writing back the write-sets to a durable and fault-tolerant storage. Both components communicate through the network, and stand in between the client and the underlying database. Execution order of client requests in DistributedLog is guaranteed, replicas execute requests in FIFO order.

Write order is guaranteed within a client session, but no guarantees for read order are provided. If a client executes a write operation and then a read operation, there is no guarantee that state changes caused by the previous write operation will be in effect in the following read operation. The state of DBMS instances may differ between replicas, but eventually they will be consistent (i.e., they will reach the same state).

The replication middleware works as follows. Client requests (i.e. those involving write operations) arrive at the distributed log and are separated into shards. The concept of sharding is explained in detail in 3.1.2. Since for the evaluation of the use cases a benchmark that mimics the management of warehouses was used, each shard corresponds to a subset of warehouses (e.g, warehouses 1 through 5). Each log instance of the distributed log contains the requests for one and only one shard. In other words, a log instance is equivalent to a shard. Each replica is responsible for processing the requests of a set of shards. For example, replica 1 is responsible for processing requests that are allocated into shard 1 and shard 2. Once a request has been executed, the state changes derived from the request are sent to the replication log. Other replicas will then read those changes and apply them to their

own DBMS instances. All communication associated to the replication mechanism is done through the replication log, replicas do not establish direct communication. Pure read operations are handled by directly connecting to one of the DBMS instances.

It is important to note that replicating a request implies less work for the system than executing it. If this weren't the case than the middleware would pose no advantage over active replication.

3.2.1 *Middleware throughput optimization*

Read operations are processed synchronously. When a client executes a read operation, a connection is created to one of the DBMS instances and the result of the read operation is returned to the client. Write operations, however, which we refer to here as **client requests**, are executed asynchronously. As soon as a client request is persisted by the distributed log, the call is returned. This means that client requests may accumulate in Distributed Log, creating a "backlog", as depicted in Figure 11. Depending on the workload and on how fast each replica can process it's shard's requests, the backlog may extend or shrink, impacting the middleware's throughput.

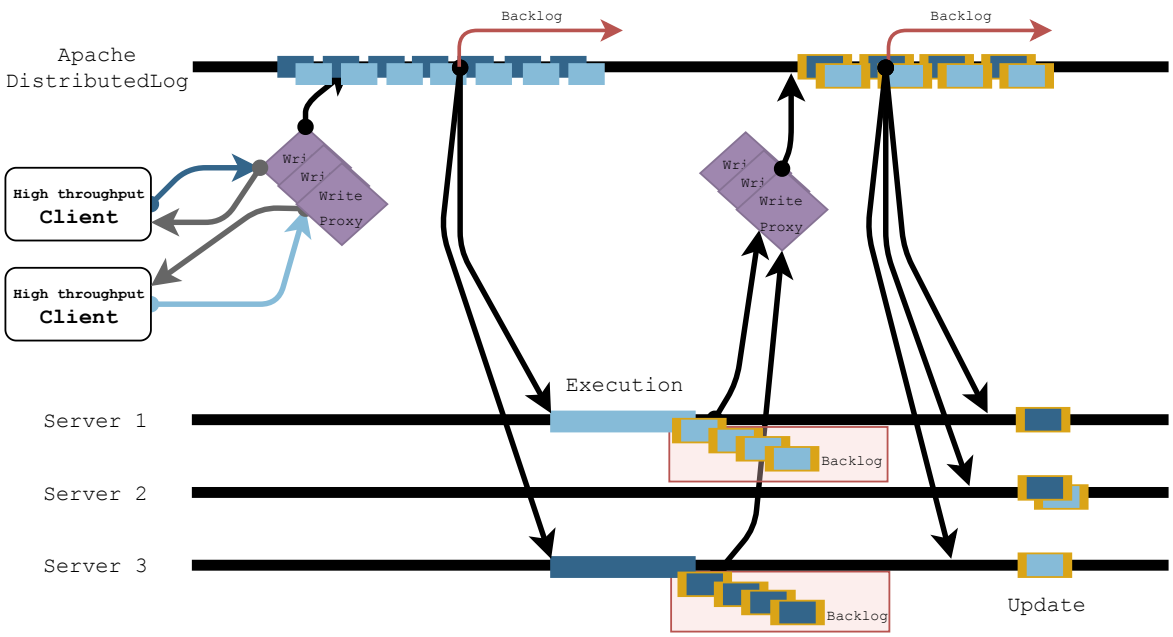


Figure 11: Replication middleware backlog representation.

Replication of client requests among replicas is also done asynchronously. There is a delay between the execution of a request and when the state changes that occur from that request are written to the replication log. There is also a delay between when those changes are written to the replication log and when they are read by the other replicas.

The throughput optimisation concerns the speed at which each replica processes client requests, and the speed at which it is able to write to and read from the replication log. That is, the optimisation concerns each replica's client requests throughput, replication log write throughput and replication log read throughput. Ultimately, the optimisation focuses in minimizing the "backlog effect" in the distributed log and the drift between replicas. It is worth noting that replicas may limit the

performance of each other when it comes to replication. Replicas can't read faster from the replication log than at the speed at which other replicas are writing to it. This means that it is important to achieve a balance between write and read speeds within the cluster when it comes to replication.

3.3 Added features

In order to support the use of RL agents, two new components are now included in the middleware: the monitoring and update systems. The monitoring system adds metric monitoring to the middleware, while the update system allows for a subset of the middleware's configuration parameters to be updated from a remote location without incurring in an interruption of requests processing or restart of the middleware or underlying databases.

3.3.1 *Monitoring system*

One of the major elements in a RL algorithm is the environment's state (see section 2.2.1), which must be measured on every step the agent takes. Since the original middleware did not offer such usage statistics, a monitoring system was implemented.

This system makes use of zeromq's publish-subscribe system [13]. The Publish/Subscribe messaging pattern offers a series of advantages among which we have scalability. If we were to scale to a cluster with a big number of replicas, the publish-subscribe pattern could prove to be an advantage.

One very important decision that has to be made regarding the monitoring system is the interval at which metrics are recorded. In a lot of cases the RL agent's environment, and thus its state changes and reward, is simulated. For example, if we make an agent whose objective is to play chess, we can simulate the opponent's move by choosing an action at random, or by using a second agent to represent that player, etc. When the environment is simulated by the agent itself, our learning speed is constrained by the computing capabilities of the machine on which the agent is running. On our case, however, we actually have to run a workload on our middleware in order to observe how the state changes, and how our reward varies. This means the speed at which our agent is able to perform a step is not limited by the computing power of the node where the agent is running, but by how long we take to measure the new state, after taking an action. When a change is made to the cluster, that change will take a certain amount of time to be reflected in the measured metrics. For instance, a given change might initially cause an observable drop in performance, but in the long run, it could lead to an increase in the overall performance of the cluster. This constrains the measuring interval to be a very important factor. If the interval is too low, the new state might not reflect the true impact of an action on the cluster. If it is too long, we'll be adjourning the time it takes to complete a step unnecessarily. But then, we also have to think about the RL agent's gamma value (consult section 2.2.1). The agent looks not only for immediate reward but also for future reward, so even if performance gains are not completely reflected in the metric measurements and immediate reward value, it could reflect in future reward values, thus allowing for shorter time intervals between metric readings. In essence, finding the optimal time interval is a very difficult task, but one that can greatly reduce the agent's training time. During our evaluation tests, the time

interval between metric readings was chosen by intuition. We leave to future work the testing of different time intervals and evaluating the impact they have on the agents' performance.

Lets take a look at the individual components, depicted in Figure 12, that build the monitoring system.

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 [13], that periodically pushes readings into the Monitor Handler components. In practice, this component feeds the reinforcement learning algorithm with state and reward updates. Updates are sent in a configurable time interval, that defaults to 15 seconds. The metrics measured for each replica are described in table 2.

Table 2: Set of metrics measured by the monitoring system.

*Hardware metrics where only implemented during the second use-case.

Configuration	Description
ClientRequests	Executed client requests.
Requests Written	Client Requests sent to replication log.
Replicated Requests	Requests read from replication log and applied to the DBMS.
NewClientRequests	Nr. of new client requests for this replica.
CommittedVirtualMemorySize*	Virtual memory available to the java process, in bytes
TotalSwapSpaceSize*	Size of swap
FreeSwapSpaceSize*	Free swap space
SystemCpuLoad*	Overall CPU load
ProcessCpuTime*	CPU time consumed by the JVM
ProcessCpuLoad*	"Recent cpu load" caused by the JVM
TotalPhysicalMemorySize*	Size of RAM
FreePhysicalMemorySize*	Available RAM

These metrics represent hardware usage and the most important data flows in the system. The configuration *ClientRequests* describes the number of client requests executed, by the replica, over the chosen time interval. *Requests Written* is the number of requests written to the replication log. Similarly, *Replicated Requests* is the number of requests read from the replication log. Finally, *NewClientRequests* is the number of new client requests that have arrived at *DistributedLog* for which the replica is responsible.

Monitor Handler

The Monitor handler collects the metrics sent by the Monitor agents in each replica by subscribing their updates through the publish-subscribe service. This data is then organized and processed to determine the current state of the environment and the reward associated with the last taken action. The Monitor Handler component is deployed outside the replica nodes, in each agent.

3.3.2 *Updates system*

Changes to the replica's tuning parameters are handled by the *Update Dispatcher* and *Update Handler* components. Both components introduce the possibility to change and adjust tuning parameters *online*, that is, without having to decommission and recommission the activities of the database engine or the middleware.

Update Dispatcher

There is one *Update Dispatcher* per each replica's agent. The *Update Dispatcher* establishes connections to the components that make up the *Update Handler* and relays to them the actions the agent decides on.

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. The Update Handler is composed by a set of instances that dynamically adjust the parameter values without having to restart the middleware or the DBMS engine. By dynamic change we mean that these parameters may be altered multiple times, while the middleware is running, with minimal overhead.

Below are the sub-components that make up the Update Handler.

Adjustable Connection Pool - This pool holds instances of database connections in a queue. The initial size of the pool is passed as an argument. Instances take connections from this pool and later release them so that they can be used by other instances. This queue is thread-safe and offers methods that allow for a calling instance to block until a connection is available, in case all the connections are being used at the time the call is made. The pool keeps a thread listening for communications on a predetermined port (also passed as an argument). The replica's agent establishes contact with this port whenever it wants to change the size of the pool. To change the size of the pool, connections are simply opened and added to the queue or taken from the queue and closed. If one connection is to be closed but all connections are being used, the pool will wait until one of them is released back to the queue before closing it.

Adjustable Executor Service - This service is essentially a pool of threads used to execute tasks. It makes use of *Workers*. The initial size of the pool is passed as an argument. The service instantiates a number of workers equal to the size of the pool. The workers share a queue and each worker runs in its own thread. The job of a worker is to take tasks from the shared queue and execute them. When the pool size is increased, new workers are instantiated and given a new thread to run on. When the pool size decreases, workers are chosen at random to be closed. The workers have a *close* variable which is set to true when the worker is to be closed. If the worker is currently executing a task, he will wait until the task is completed before closing.

Adjustable Integer - An integer value that can be changed remotely. It waits for connections on a predetermined port, so that changes to the integer value are received through the established sockets. The integer is thread-safe, meaning that concurrent calls to consult the integer's value and to change it do not cause problems.

3.3.3 *Tuning parameters*

The set of tuning parameters considered in this approach to sponsor the dynamic adjustment of parameters is presented in Table 3. They can now be updated at any time while the middleware is running without incurring in an interruption of the middleware's normal operation.

Table 3: 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 of writes to the replication log
db.writeset.delay	Delay between writes to the replication log

The *db.pool* parameter is regulated using an *Adjustable Connection Pool*. *dlog.reader.threads*, in turn, uses an instance of the *Adjustable Executor Service*. Finally, both *db.writeset.transactions* and *db.writeset.delay* are controlled using *Adjustable Integers*.

db.pool is the number of connections in the pool that is used by a replica to communicate with it's underlying DBMS. More specifically, the connections in this pool are used to apply changes read from the replication log to the underlying DBMS instance of the replica.

dlog.reader.threads is the number of active threads applying replicated changes to the underlying DBMS instance. Each replicated client request is attributed to an available thread, blocking if there isn't one. Then, the thread must first acquire a fair lock that ensures there is only one thread working on non-independent sections of data. The fairness property of the lock is what ensures the locks are acquired in a FIFO order, so that the write order is maintained. After that, it must acquire a database connection from the pool (see db.pool above), and apply the replicated transaction using that connection.

db.wriset.transactions sets the maximum number of client requests that can be added to the replication log, during a single write. In other words, it's the maximum batch size of each write operation.

db.writeset.delay sets the time delay between writes to the replication log.

In summary, this dissertation will analyze the optimization of the replication middleware here presented, to which the dissertation's author has made prior contributions, see [11]. Besides those, in this dissertation, monitoring and updates systems were added to the middleware. These both complement the features of the middleware as well as allow for it to be tuned using DRL techniques. Although the middleware has other configuration variables, for the purposes of this dissertation the variables **db.pool**, **dlog.reader.threads**, **db.wriset.transactions** and **db.writeset.delay** were selected to be configured by the replica level agent, since they provide the highest impact on the middleware's performance.

4

TUNERL

Chapter 4 introduces TuneRL, an automatic parameter tuning system tailored for distributed databases, that considers deep reinforcement learning strategies as the underlying optimisation mechanism. TuneRL is designed and prototyped as a database middleware optimisation system, highlighting the pluggable nature of the design and allowing it to be used with other database systems from a design point of view.

Two use cases were considered, promoting an applied implementation of the system. The use-cases focused on tuning the parameters of an experimental database replication middleware system, allowing to depict the application and convey its evaluation.

The first use case consists of a RL agent capable of tuning individual parameters of the cluster's replicas - Replica Agent.

The second one is a mechanism capable of migrating shards between replicas based on a RL agent that decides when to migrate a shard from one replica to another - Shard Agent.

In both cases, the objective is to optimize the overall throughput of the database cluster.

4.1 TUNERL ARCHITECTURE

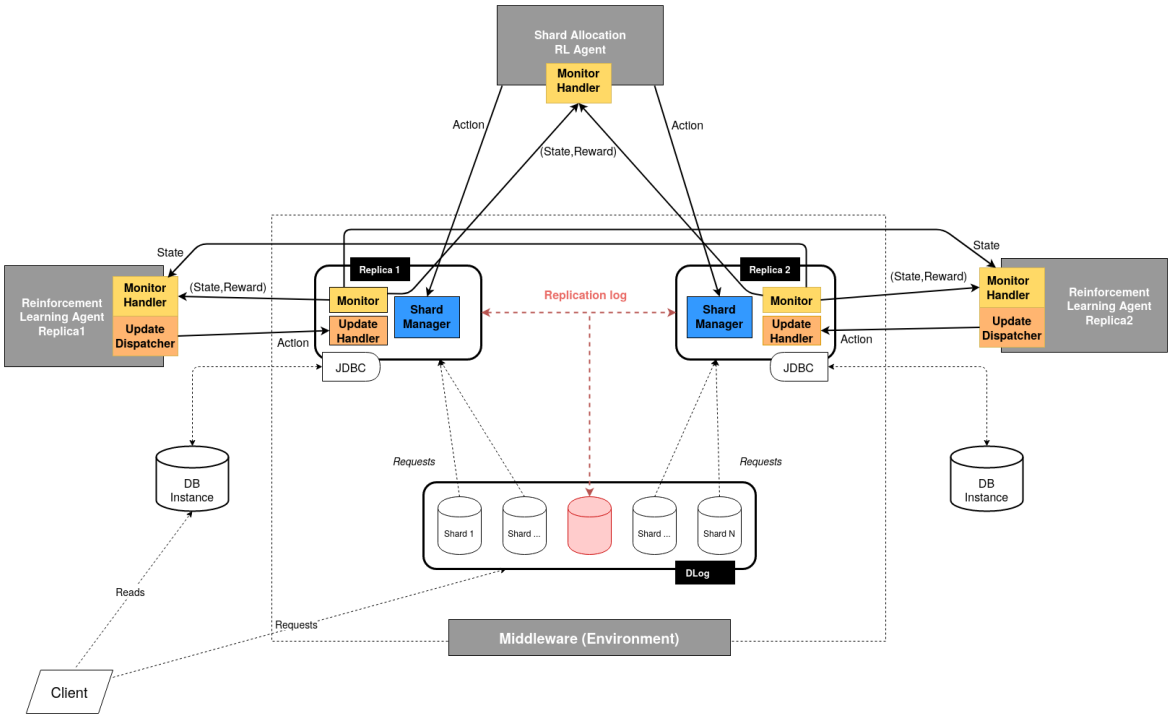


Figure 12: TuneRL architecture.

Figure 12 depicts both the middleware and all the elements that were added to it for the RL systems. Dotted arrows are used to represent the interactions of the original middleware within the system. Solid arrows are used to represent new interactions occurring from the newly implemented solutions.

We can see that each replica has its own replica level RL agent, but there is only one shard RL agent managing all the replicas. The environment for both agents, the middleware, is delimited by the big dotted square. On the lower side of the environment is DistributedLog, with the log instance used for replication highlighted in red. On the upper side are the replicas.

Each replica contains a Monitor, an Update Handler, a Shard Manager and a JDBC interface. Each replica communicates with the underlying DBMS instance through the JDBC provided by the vendor. The monitor component publishes values of the monitoring metrics in periodic intervals, using ZeroMQ [13]. The UpdateHandler is an abstraction of all the class instances used to apply the actions issued by the agents. The ShardManager is responsible for receiving orders from the shards RL agent, as well as shard transfer orders coming from the replication log, and taking all the steps necessary to close/open shards within a replica, while guaranteeing consistency within the cluster. On each replica RL agent and on the shard RL agent there is a Monitor Handler and an Update Dispatcher. The Monitor Handler subscribes the information published by the Monitor components of the replicas. The Update Dispatcher relays actions from the agents to the replicas through communication sockets.

4.2 RL AGENTS OVERVIEW

The RL agents built for both use cases use DQL (2.2.3) as their algorithm. While this is not the most advanced alogrithm, the fact that it can only deal with a finite amount of actions is considered benefitial for the use cases here presented. By discretizing the action space, see 4.3.2, only incremental changes can be made on each step, thus making the training more stable in terms of performance variation between steps. The evaluation of alternative algorithms is left to future work.

The agents were built using keras [14] and OpenAI gym [6]. Keras implements the necessary components for the neural networks used by the agents. On the other hand, OpenAI Gym "is a toolkit for developing and comparing reinforcement learning algorithms". Essentially it let's us build gym environments (as in "an agent's environment").

In a gym environment, we only have to set *init*, *step* and *reset* functions. The *init* should take care of any initialization steps, such as instantiating state variables. The *reset* is called whenever an episode ends and a new one starts, so that clean-up steps may be taken. Most importantly, it is in the *step* function that most of the logic is coded. It receives the action chosen by the network as an argument, and must apply that action to the (RL) environment as well as retrieve the new state and calculate the reward associated with the undertaken action.

The environment is then passed as an argument to an instance of Kera's DQNAgent, an implementation of DQL (the chosen algorithm for the use cases), together with a pre-specified neural network built with keras and other parameters in order to fit (i.e., train) our agent.

The number of input nodes of the neural network must match the environment's number of state variables, and the number of output nodes must match the number of possible actions. The knowledge gathered by the agent is stored in the weights of the neural network.

We used the epsilon greedy algorithm in our policy. Actions are chosen at random according to a value epsilon which varies between 0 and 1. If epsilon is set to, for example, 0.1 then actions will be chosen at random 10% of the time, otherwise the best action outputted by the neural network is chosen. This allows for the algorithm to try unexplored options in case they are actually better than the action the neural network chooses from previous knowledge, avoiding getting stuck on local optimums (exploitation vs exploration). The gamma parameter (see 2.2.1) is set to the default value, 0.99. This means that future reward is valued as much as immediate reward. The optimization function (i.e., the function used to update the neural network's weights) chosen was Adam [15]. Also, some optimization options offered by keras were activated, namely *double dql* [28] and *dueling network* [29]. Double dql essentially avoids the overestimation of q-values by adding a second neural network that evaluates the actions the main neural network chooses. Dueling network allows for a better policy evaluation, and consequent adjustment, when multiple actions have similar results. Testing other combinations of parameters is left to future work.

4.3 REPLICA AGENT

Each replica's agent, is responsible for tuning a set of parameters that belong to its assigned replica, the agent's **environment**. The idea here is to tune each replica's individual behaviour, which will consequently affect the overall cluster performance.

This agent makes use of a set of subcomponents: the Monitoring components (see section 3.3.1), the Update Handler, and the Update Dispatcher. The Monitor probes and aggregates metrics about each replica that are afterwards pushed to the Monitor Handler. This metrics comprehend the **state** and **reward** of our RL system. The state and reward 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. The changes applied to the environment are the agent's **Actions**. Changes are applied online, without having to restart the middleware or the underlying DBMS instances. The combination of possible states, for this use case, represents a large search space. Some state variables are continuous, so a simpler algorithm, such as Q-Learning, would not be suitable. For this reason, DQL is the selected candidate as the base for the agent's policy.

4.3.1 *States*

The states taken into account by the RL method are a composition of two sets of variables. The first is the set of current values of the parameters that are being optimized, depicted in Table 3. The second is a set of metrics that characterize the overall performance of a 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 is depicted in Table 4.

Table 4: Set of elements that compose the state of replica agents.

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 of writes to the replication log
db.writeset.delay	Current delay between writes to replication log
ClientRequests	Executed client requests
Requests Written	Client Requests sent to replication log
Replicated Requests	Requests read from replication log and applied to the DBMS
NewClientRequests	Nr. of new client requests for this replica
rep_Txn in DL	Client Requests sent to replication log by other replicas

The metrics are passed to the agent as an average, in ClientRequests/second, since the last state update. *NewClientRequests* and *ClientRequests* allows to establish a relationship between the number of requests received by DistributedLog that must be processed by the replica and the number of requests that were actually executed. Moreover, the write speed to the replication log of other replicas of the system is 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 requests is due to a lack of performance from the replica itself or because there aren't a lot of requests in the log to replicate.

4.3.2 *Actions*

The set of actions builds the possible changes that can be made on each step to the environment. While using Deep Q-Learning, actions must be discrete (2.2.3). For this reason, actions are sampled in the form of incremental changes. This allows not just to make them discrete but also to only slightly change the settings on each step, thus avoiding consequences that could arise from making harsh changes to the environment in a single step, especially in the initial stages of learning. A possible alternative would be DDPG. In theory DDPG would be able to set all variables to any value on each step. We leave to future work evaluating whether this algorithm would be a better alternative. The possible set of actions is depicted in Table 3.

Table 5: Actions considered for replica agents.

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 tx.
Decrement db.writeset.transactions	Decrement by 100 tx.
Set db.writeset.transactions special	Set to 0.
Increment db.writeset.delay	Increment by 100 ms.
Decrement db.writeset.delay	Decrement by 100 ms.

In order to prevent system crashes due to changes to an out-of-bound configuration of some parameters, an upper and lower limit is set for each tuning variable. The increment and decrement values were chosen by trial and error. The objective is set 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 becomes very low (taking into account the boundaries set). The configuration variable *db.writeset.transactions* can be set to a special value of 0. When this value is set, the limit on how many transactions can be written on each batch is removed.

4.3.3 *Reward*

The main goal is to optimize the cluster's overall throughput, as explained in 3.2.1, which means that a higher throughput represents a better outcome. Consequently, the reward function is associated with the throughput, being defined as the sum of all the latest metric averages that refer to replica throughput. Since all the averages are in client requests 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.

$$reward = ClientRequests + RequestsWritten + ReplicatedRequests$$

4.3.4 *Neural network*

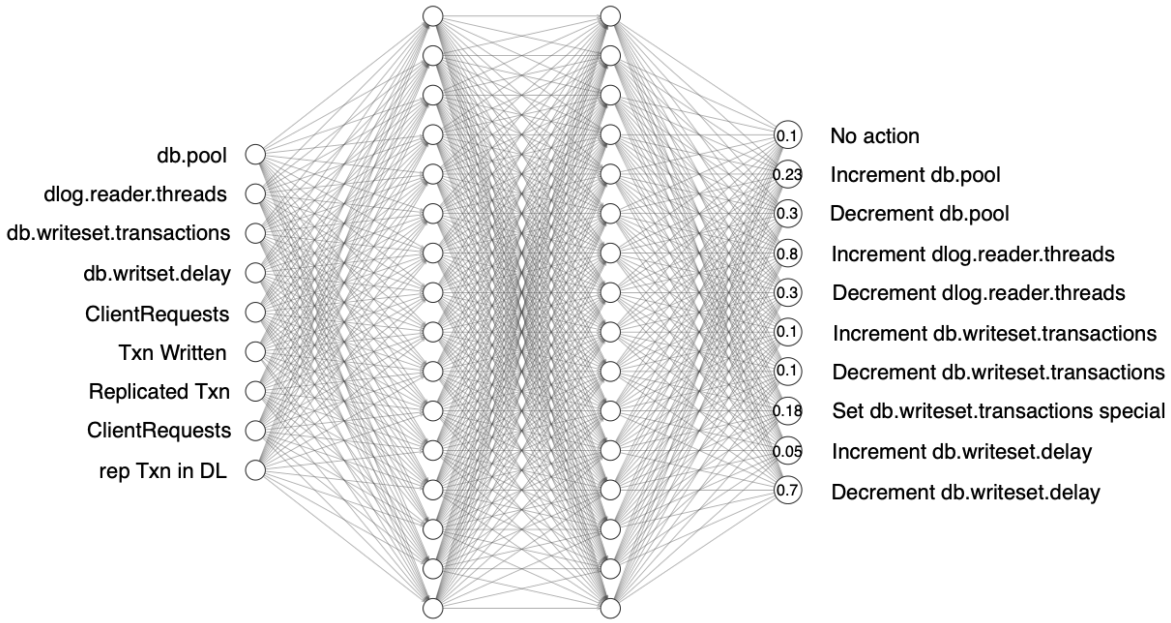


Figure 13: Replica agent's decision making neural network.

Figure 13 depicts the neural network used to map states to actions in replica agents. It has two 16 node hidden layers. The number of nodes and intermediate layers was chosen by trial-and-error. Each input node corresponds to one of the state variables described in Table 4, and each output node corresponds to the q-value of an action for a given state. Actions are described in Table 5. For the example shown in Figure 13 the chosen action would be "Increment dlog.reader.threads", since it's node has the highest q-value.

4.4 SHARD MANAGEMENT AGENT

The shard management agent represents the second use case of dynamic parameter tuning. It manages the allocation of shards by reassigning them among the existing replicas. The idea is that different replicas may have different hardware capabilities. Thus, the system may benefit from an uneven distribution of shards. To fulfil the use case, a reallocation mechanism was designed and integrated into the foreground replication middleware. For the purpose of implementing the reallocation mechanism, it is assumed that the agent knows at all times which replica owns each shard and that orders released by the agent are always correct.

4.4.1 *Shard reallocation*

The middleware was restructured so that it now has three managing classes: **Client Requests Manager**, **Replication Manager** and **Shards Manager**. The *Client Requests Manager* holds all instances

of services that are related to processing Client Requests. In turn, the *Replication Manager* holds all instances that deal with replication of transactions, be it reading from or writing to the replication log. Finally, the Shards Manager handles reallocation requests from the agent and replication log, communicating with the Client Request and Replication Managers in order to stop or start processing shards that are being reallocated. The diagrams from figure 14 to figure 17 detail the reallocation process of a shard named *S1* to a replica *R2*.

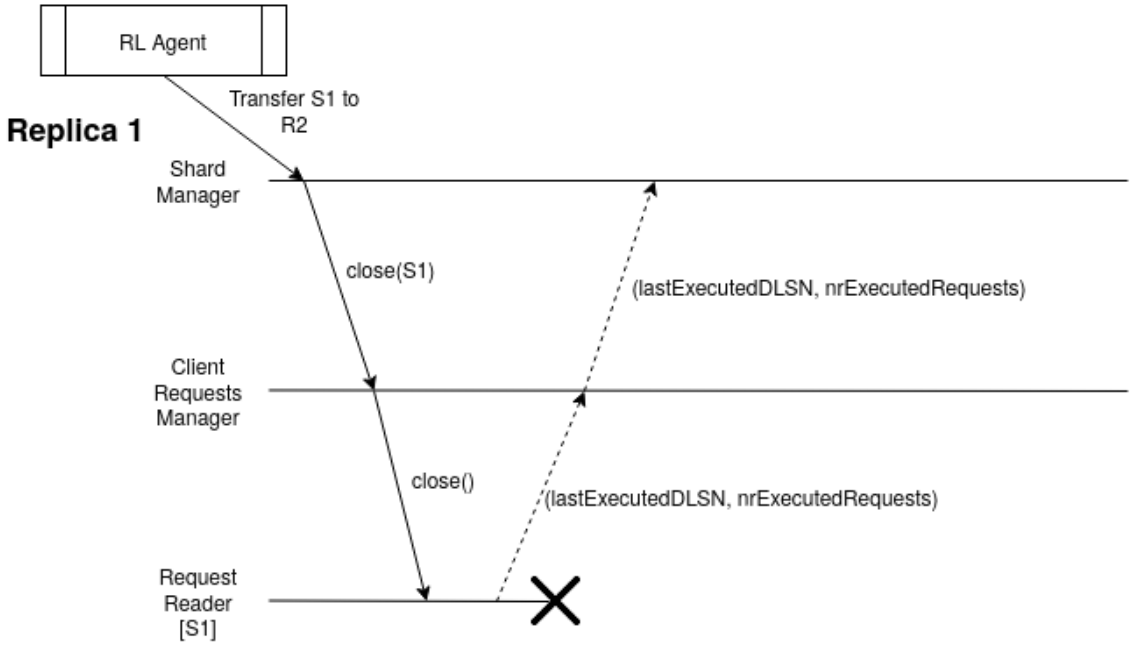


Figure 14: Shard Agent explained: Agent emits order; replica stops processing requests from shard S1.

In Figure 14, the agent emits the order to transfer the shard. It keeps track of which replica owns a given shard, so it knows that replica *R1* is the one holding *S1*. The RL agent will contact this replica's Shard Manager, communicating it's intent to move this shard over to replica *R2*. The shard manager will then take over the reallocation process. It first communicates with the Client Requests Manager, indicating it should close the request reader that is reading requests from *S1*. From the moment the request reader is closed, replica *R1* is no longer processing requests from *S1*. However, it must finish outstanding replication tasks, replicating the ones it has already executed and inform the next replica of the last record it has read from DistributedLog. For this reason, when the request reader is closed, it returns the number of requests it has processed and the last processed *DistributedLog Sequence Number (DLSN)*.

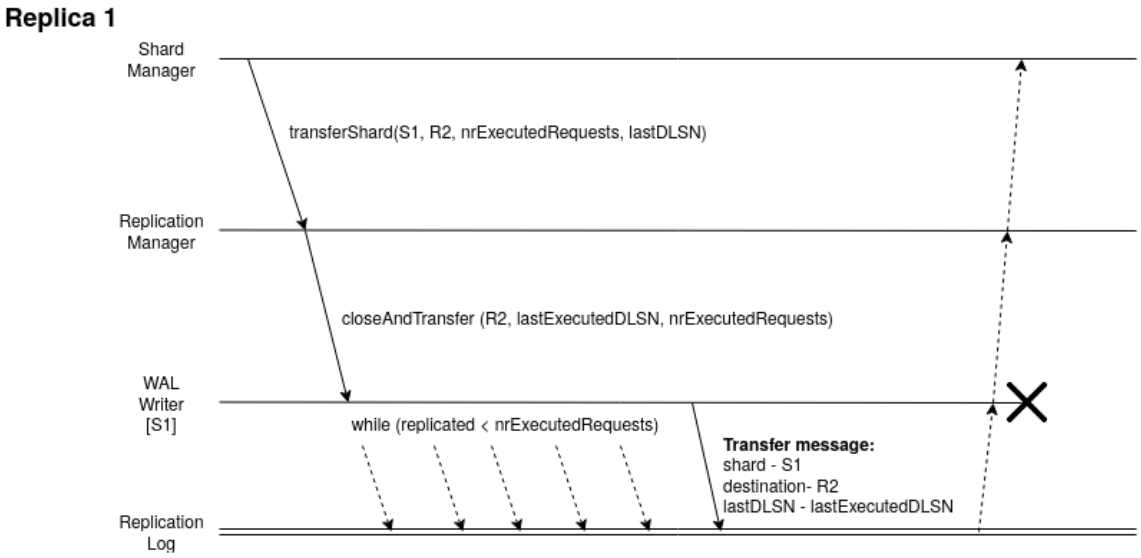


Figure 15: Shard Agent explained: Replica stops replicating requests from shard S1; replica writes transfer message to replication log.

In Figure 15, after communicating with the Client Requests Manager, the Shard Manager must now command the Replication Manager to close the writer that is responsible for replicating *S1* to the Replication Log. The Replication Manager passes on the number of executed Client Requests over to the *Write Ahead Log (WAL)* writer. Looking at the number of processed Client Requests, the writer will wait until it has replicated all of those requests before closing. Once it has replicated all requests, it writes a special message to the replication log, saying that *S1* is to be transferred to replica *R2*, and that the last DLSN this replica has processed is *lastExecutedDLSN*.

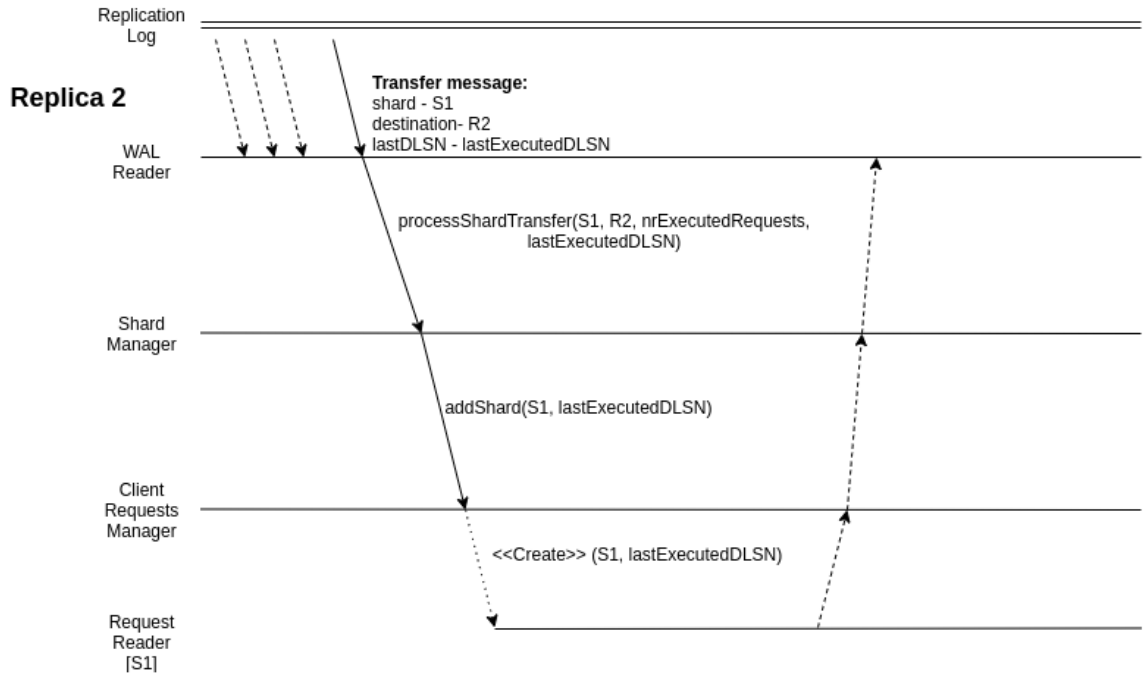


Figure 16: Shard Agent explained: Replica reads transfer message; replica opens request reader.

The Shard takeover process is depicted in Figure 16 and Figure 17. In Figure 16, on replica *R2*'s side, there is a reader processing transactions from the Replication Log. As the message stating that there is a shard destined to this replica is collected from the Replication Log, it is passed on to the Shard Manager. The Shard Manager instructs the Client Requests Manager to add shard *S1*. The Client Request Manager creates a Request Reader dedicated to cater to the needs of the new shard.

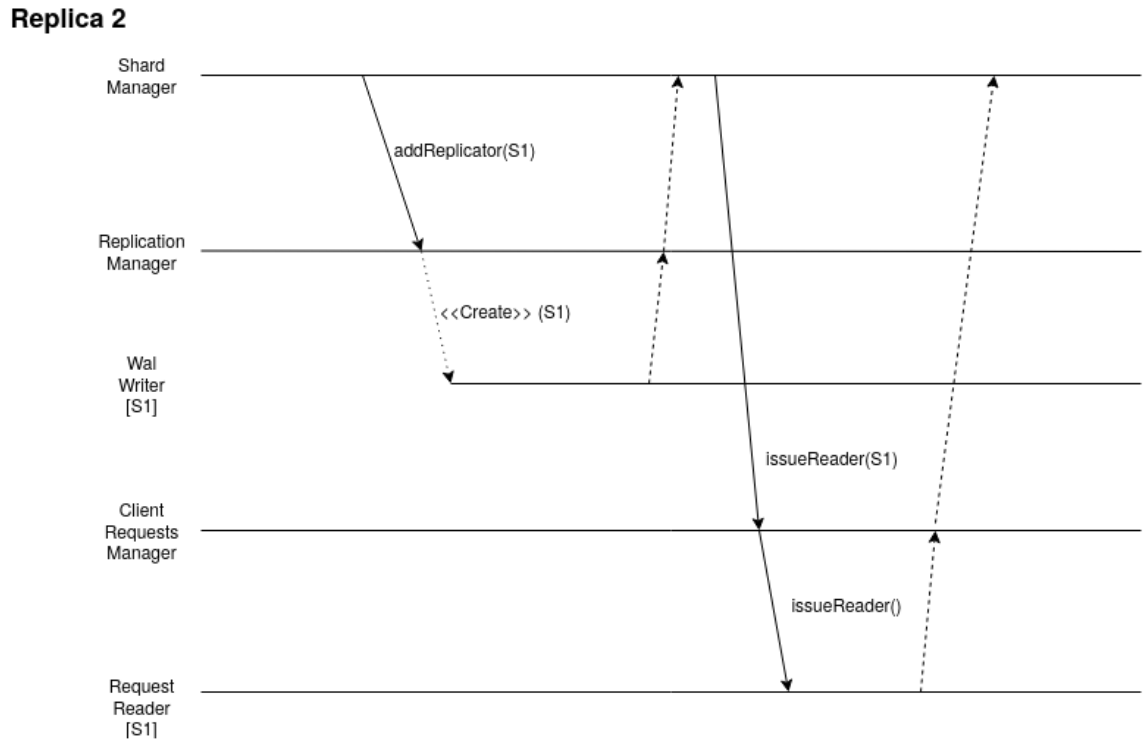


Figure 17: Shard Agent explained: Replica creates and starts shard replicator; replica starts request reader.

The process description continues in Figure 17. The Shard Manager passes on to the Replication Manager to start a new WAL writer that will replicate requests that will be processed from *S1*, to the replication log. Finally, the Shard Manager commands the Request Reader to start processing requests from the new shard. The Reader has access to the last processed DLSN from replica *R1*, and so starts executing requests starting at the record immediately after. The reallocation process is then complete.

4.4.2 *State*

The state of the shard allocating RL agent is composed by most of the same variables as the tuning parameters RL agent, but here we added a set of hardware metrics to complement that state, that are offered by an available API (Java OperatingSystemMXBean). The full set of variables for the shard allocator agent is represented in Table 6.

Table 6: Set of elements that compose the state in the shard allocation RL process.

Configuration	Description
ClientRequests-Rx	Executed client requests
Requests Written-Rx	Requests sent to replication log
Replicated Requests-Rx	Requests read from replication log and applied to the DBMS
NewClientRequests-Rx	Nr. of new client requests in DistributedLog
CommittedVirtualMemorySize-Rx	Virtual memory available to the java process, in bytes
TotalSwapSpaceSize-Rx	Size of swap
FreeSwapSpaceSize-Rx	Free swap space
SystemCpuLoad-Rx	Overall CPU load
ProcessCpuTime-Rx	CPU time consumed by the JVM
ProcessCpuLoad-Rx	”Recent cpu load” caused by the JVM
TotalPhysicalMemorySize-Rx	Size of RAM
FreePhysicalMemorySize-Rx	Available RAM
NrOwnedShards-Rx	Number of shards assigned to replica Rx

Since, contrary to the agent in the first use case, the shard reallocator manages information from all replicas, there is a set of variables for each replica in the cluster, described in Table 6 by the designation *Rx*. So, the number of variables in our state will be equal to $13 * N$ where N is the number of replicas in the cluster.

4.4.3 *Actions*

There is a very simple set of actions the agent can take. For every pair of replicas in the cluster, on each step, the agent can only take one shard from one of those replicas and assign it to another. The shard to be conceded is chosen at random from the set of shards that the conceding replica owns. For example if there are two replicas R1 and R2, the agent may only take one shard from R1 and assign it to R2 or vice-versa.

Table 7: Actions considered in the shard allocation RL process.

No action	Shard allocations are kept unchanged
Increase Rx; Decrease Ry	Take one shard from replica Ry and reallocate it to replica Rx

4.4.4 *Reward*

Similarly to the agent in the first use case, the shard reallocation RL agent uses as a reward function the sum of all client requests processed, replicated from the replication log and written to the replication log, by all replicas.

$$reward = sum(ClientRequests_{Rx}) + sum(TxnWritten_{Rx}) + sum(ReplicatedTxn_{Rx})$$

4.4.5 *Neural Network*

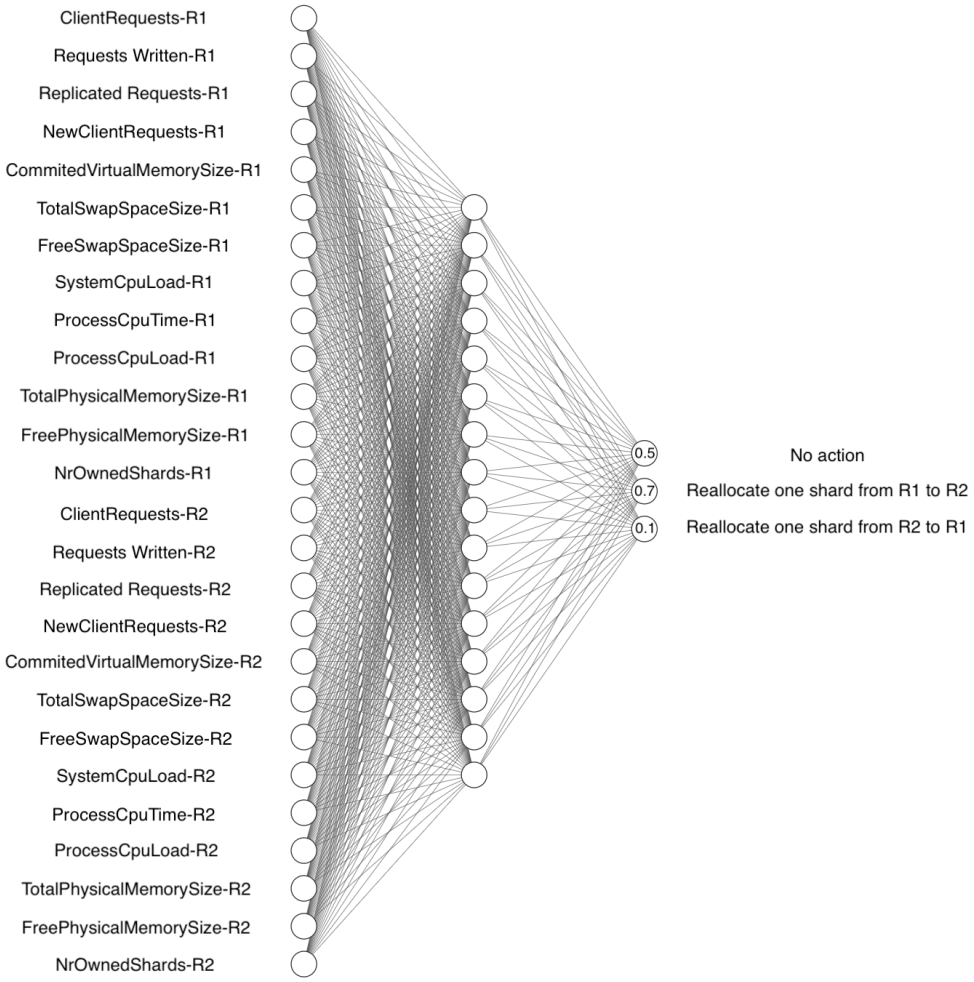


Figure 18: Shard agent's decision making neural network.

The shard agent's neural network has only one hidden layer with 16 nodes, contrary to the 2 layers in the replica agents 4.3.4. The process of selecting the network's architecture is the target of much debate within the developers of the field. Online research shows it is general consensus that the maximum number of hidden layers one needs to use, save for some very complex applications, is 2. But, 1 suffices in most instances. For example, see [21]. From the rules of thumb also proposed in [21] "The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer", which would equate to 20 neurons, but since 16 neuron layers were effective for the first use case, that was the selected number of hidden neurons (1/2 the size of the input layer, plus the size of the output layer). Each node of the input layer represents one of the state variables from Table 6 and each output node equates to the q-value of one of the actions from Table 7. In the example shown in Figure 18 the chosen action would be to "reallocate one shard from R1 to R2" since it's the one with the highest q-value.

4.5 SHARD MANAGEMENT AGENT FAULT MODEL

Due to the distributed and multi-threaded nature of the middleware, there are certain complexities that could cause the occurrence of faults during the reallocation process. Special mechanisms were developed to mitigate the occurrence of these faults. Below is a list of all the identified potential faults that were dealt with.

Fault 1: Transfer a shard the replica doesn't own

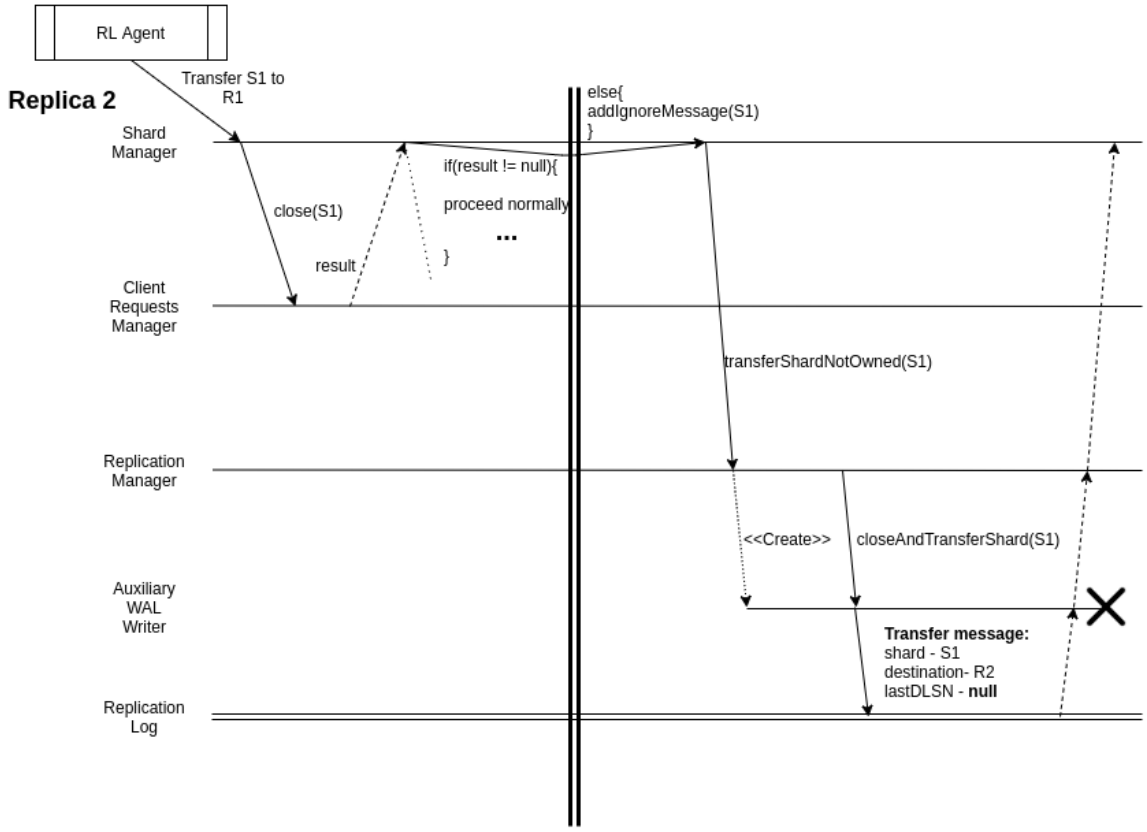


Figure 19: Shard Agent Fault Model. Fault 1: Transfer a shard not owned by the replica - Ignore message mechanism.

Consider the scenario depicted in Figure 19. The agent instructs replica *R1* to transfer *S1* to replica *R2*. After that, the agent instructs replica *R2* to return *S1* to replica *R1*. In this instance, if *R2* hasn't yet received *S1* when it receives the instruction to return it to *R1*, two outcomes are possible. In the worst case scenario, it will cause the replica to crash. Otherwise, we could make the Shard Manager simply ignore the agent's order, attributing it to an error on the agent's part, but it is assumed that the agent is always right, so that is not possible. The solution opted for uses an Ignore Message mechanism. When the replica receives an instruction to transfer a shard, it first checks whether it already owns that shard or not. If it doesn't, and in accordance with the assumption that all messages

sent by the agent are correct, it assumes that the shard is on route to this replica. The replica then creates an Ignore Message, stating that when it eventually receives a message related to *S1* it can safely ignore it. After that, in order to not delay the process of returning *S1*, it immediately writes to the end of the replication log a transfer message, registering the transfer of *S1* back to *R1*.

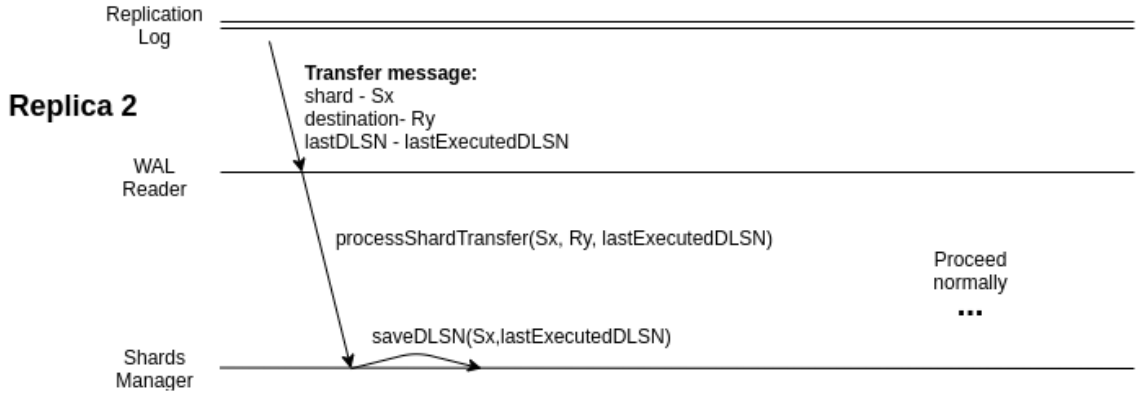


Figure 20: Shard Agent Fault Model. Fault 1: Transfer a shard not owned by the replica - Save DLSN values.

Aside from that, as depicted in Figure 20, all replicas now have to store the last processed DLSN of each shard transfer message that passes through them, because in the cases where an Ignore Message is created, the associated message that is written to the replication log will not have the last processed DLSN. So, if a replica receives a shard transfer message without the DLSN, the replica will have to turn to this table to know at what point of the log it should start processing requests from. The Ignore Message mechanism makes it so that even if replica *R1* is behind on processing replicated requests, it won't slow down the shard transfer process, making the reallocation system and the agent more efficient. The agent is now able to reason that *R2* is unresponsive and not processing any requests from *S1* for a given period, so it renders as the better option to return that shard to *R1* (or to any other replica). If this replica is delayed to a point that it receives multiple messages to take over a certain shard and to then return it, it simply stacks Ignore Messages on top of each other. This way the system never has to wait for a replica to catch up in order to move shards.

Fault 2: Transfer a shard and receive that shard from another replica at the same time

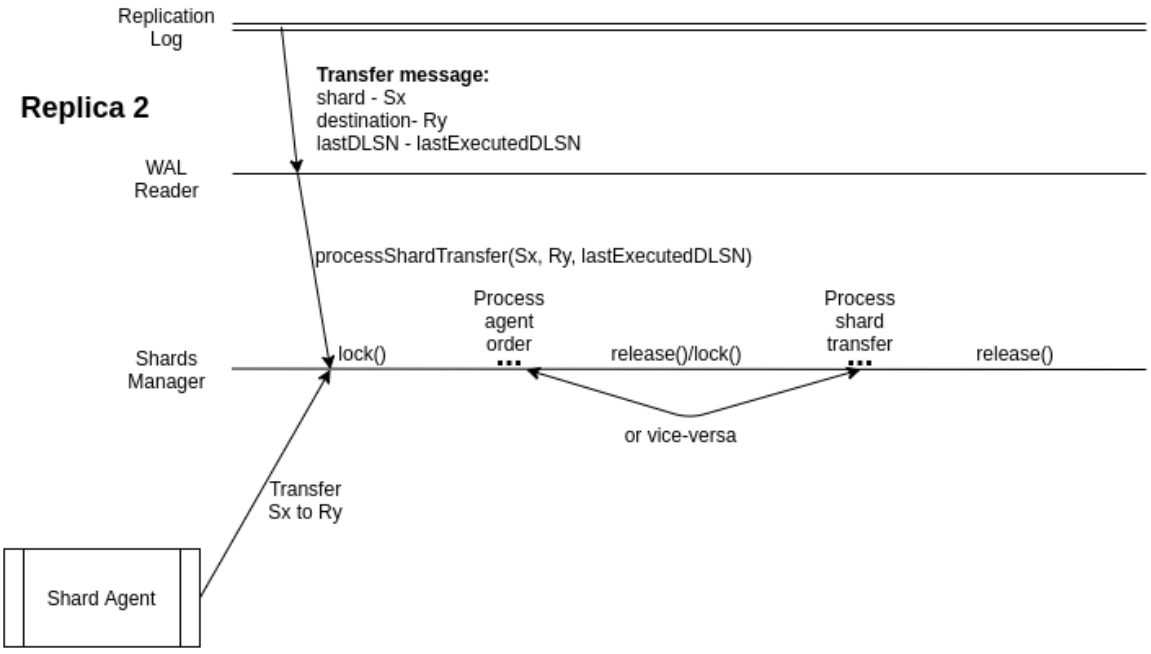


Figure 21: Shard Agent Fault Model. Fault 2: Transfer a shard and receive that shard from another replica at the same time - Lock mechanism.

This fault is complementary with Fault 1. Consider that replica *R2* receives, at the same time, a message from the shard agent to transfer shard *S1*, and a message from the replication log to take over shard *S1*. The solution presented in Fault 1 allows for the take over message to be delayed and arrive after an order to move that shard to another replica occurs, but it cannot handle a situation where both messages arrive at the same time. Since the replication log reader and the socket that communicates with the agent operate on different threads, both events would be processed in parallel. This could lead to an unwanted outcome, such as the replica taking over the shard and also writing in the replication log a message to transfer it on to the next replica. In order to avoid this, a lock mechanism is considered to the Shard Manager, which essentially renders its execution to a sequential one. Now, it either processes the shard take over message or the shard agent message first, but not both at the same time.

Fault 3: Take over shard before replicating all previous requests that pertain to that shard

The order by which the requests are executed affects the final state of the database, so execution order is relevant. This is why, for example, it is important that the distributed log has ordered records. The log readers will first read the oldest records (similar to FIFO).

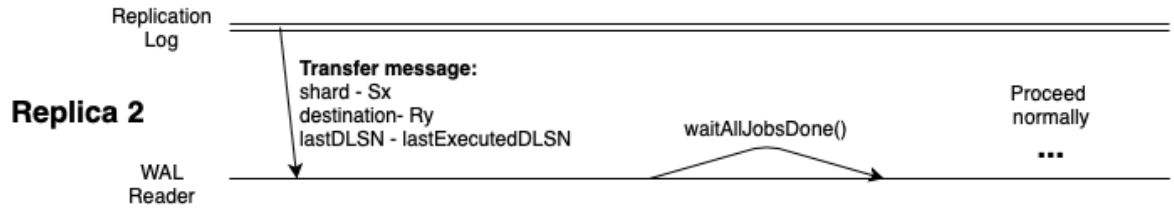


Figure 22: Shard Agent Fault Model. Fault 3: Take over shard before replicating all previous requests that pertain to that shard - WAL reader waits for all requests to finish processing.

This means that, in order for the correct state to be achieved, a replica can only start processing new requests from a shard after it has replicated all the shard's previous requests, as depicted in Figure 22. The problem is the replication log reader (WAL Reader) uses multiple threads to process the replicated requests. Execution between independent data contexts is carried out in parallel. The log reader executes requests from different shards, and even micro-shards within each shard, simultaneously. All previous requests pertaining to the shard that is being acquired will have come before the shard transfer message. In order to guarantee that all those requests have been executed, the reader temporarily stops reading new requests and waits until all the replicated requests it has read so far finish being processed. Only after all requests have finished executing does it send the message to the Shard Manager to be processed.

Fault 4: Orders from the agent being out of order in the replication log

The shard transfer messages could get out of order if the agent calls were asynchronous. The agent instructs replica *R1* to move shard *S1* to shard *R2*. Immediately after the Shard Manager receives this order, it closes the connection with the agent. From the perspective of the agent, its interaction with the replica is over, and it is free to keep on making decisions. On the replica's side however, it is executing the required takeover to transfer the shard. Despite that and, since the agent is free to take new decisions, it contacts replica *R2* and tells it to return the shard to replica *R1*. The issue here is that messages could end up in the wrong order on the replication log, if *R2*'s Shard Manager is faster than *R1* at processing the request.

From the above interaction we would get messages M1 and M2. Message M1 states that shard *S1* should now be processed by replica *R2*. Message M2 states that shard *S1* should be processed by replica *R1*. The correct order for this messages is M1 first and M2 afterwards. If the messages are inverted, message M2 will be picked up by replica *R1*. Replica *R1* will try to open a new reader for shard *S1*. It will have to turn to its DLSN records, because this message won't have a DLSN record, as it is associated to an Ignore Message on replica *R2*. However, the correct DLSN value from which to start from is stored in message M1, so it either won't be able to check the DLSN value or will retrieve an outdated one. This implies that the system can only function correctly if the messages are written to the replication log in the correct order.

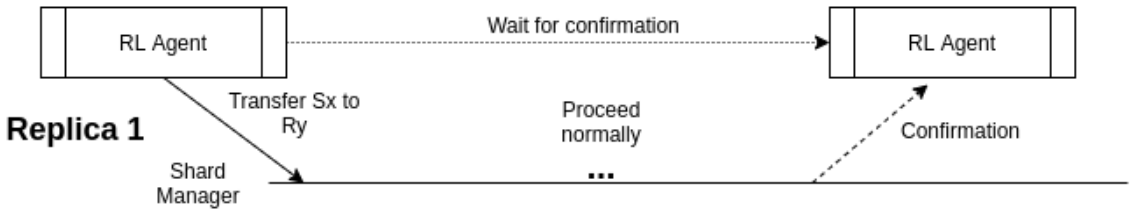


Figure 23: Shard Agent Fault Model. Fault 4: Orders from the agent being out of order in the replication log - Agent waits for confirmation before proceeding.

Therefore, the agent must be synchronous in regards to the processing of a shard transfer request by a replica's Shard Manager. Instead of immediately freeing up the agent after the request arrives at the Shard Manager, the agent will wait for a confirmation message from the Shard Manager, stating that it has written the shard transfer order to the replication log. This is depicted in Figure 23. The agent will only take new decisions after it has received that confirmation.

5

EVALUATION

The evaluation of the system was conducted considering the TPC-C benchmark [26], 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.
While benchmark runs follow the same configuration in each use-case, we could say that the workload changes during it's execution. The benchmark runs for a configured period of time, during which there is high input of new requests into DistributedLog as well as read operations that are directly executed over one of the database instances. After that time interval, there will be no more read operations and the DistributedLog will have more resources to service requests from the replicas instead of the client. At this stage the system must deal with client requests backlogged in the DistributedLog. As the backlog empties, the workload for the replicas will start shifting more towards replication, unless replication is being carried out at the same speed as client requests. So, we can distinguish at least two, possibly three, variations in overall workload that might have different optimal configurations and that the agents can try to exploit to optimize system performance.

The assessment of TuneRL was conducted on a per use-case basis, each one considering a distinct setup, but maintaining the overall goal for throughput assessment and improvement.

5.1 REPLICA AGENT

Each replica's agent has the task of optimizing it's replica throughput by modifying tuning parameters exclusive to that replica. Since the RL's algorithm uses a neural network that effectively works as a black box, there is no way to know the reasoning behind the agents' actions. So, the results analysis will be focused instead on the reward evolution over the episodes, the collected metrics values, and on the decisions taken by the agent. Remembering what is said in section 2.2.1, the agent uses it's **policy** to match the current **state** of the **environment** (the middleware) to an **action**. The agent then gets a **reward** associated with the action taken, which it uses to update it's policy. Each cycle of retrieving the state of the environment, taking an action and adjusting the policy according to the received reward is called a **step**. To a set of steps that lead to a final state, in our case processing all the requests from a benchmark run, we call an **episode** or learning cycle.

5.1.1 *Setup*

The evaluation was conducted 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 episodes 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. In other words, a set of default values for the tuning parameters was chosen, and throughout the episodes the agent incrementally modified those values in order to adapt the replica's configuration to the workload. At the beginning of each episode the values were reset to the default so that the initial state of the middleware doesn't differ between episodes.

The first 5 episodes 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 (epsilon value) of 20%.

On the final 5 episodes, 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 5. 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.

The gamma value was set to the default value of 0.99 in all episodes, so that future and immediate reward were valued equally. A minimum time interval between steps of 15 seconds was applied (metrics retrieval time interval).

For comparison, a baseline was set by running the benchmark and collecting metrics without modifying any tuning parameter during that benchmark.

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. The database was reset between each learning cycle.

5.1.2 *Results*

Table 8 depicts the results of the learning cycles of the reinforcement learning agent. The average results reported are the average results for each episode. 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 was achieved at around 81%.

Table 8: Results of replica agents' evaluation. Baseline and cycle results in client requests per second (Client Requests/sec).

Metric	Baseline	RL#1	Gain	RL#2	Gain
ClientRequests-R1	80.80	94.07	**+16.42%**	91.87	**+13.70%**
Replicated Requests-R1	35.24	55.56	**+57.64%**	55.00	**+56.05%**
Requests 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 Requests-R2	27.16	61.75	**+127.40%**	91.56	**+237.18%**
Requests 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#3	Gain	RL#4	Gain
ClientRequests-R1	80.80	91.39	**+13.11%**	99.96	**+23.71%**
Replicated Requests-R1	35.24	49.11	**+39.34%**	52.57	**+49.15%**
Requests Written-R1	27.57	89.53	**+224.78%**	99.96	**+262.59%**
ClientRequests-R2	178.20	166.63	-6.50%	142.30	-20.15
Replicated Requests-R2	27.16	60.63	**+123.27%**	99.96	**+268.09%**
Requests Written-R2	31.86	155.28	**+387.32%**	142.30	**+346.57%**
Avg Reward-R1	143.61	230.04	**+60.18%**	252.48	**+75.81%**
Avg Reward-R2	237.22	382.54	**+61.26%**	384.55	**+62.11%**

Metric	Baseline	RL#5	Gain	RL#6	Gain
ClientRequests-R1	80.80	85.87	**+6.27%**	86.04	**+6.49%**
Replicated Requests-R1	35.24	46.50	**+31.93%**	52.09	**+47.79%**
Requests Written-R1	27.57	70.44	**+155.52%**	86.04	**+212.11%**
ClientRequests-R2	178.20	181.53	**+1.87%**	207.00	**+16.16%**
Replicated Requests-R2	27.16	61.69	**+127.15%**	68.21	**+151.16%**
Requests Written-R2	31.86	72.29	**+126.85%**	111.55	**+250.09%**
Avg Reward-R1	143.61	202.81	**+41.22%**	224.17	**+56.09%**
Avg Reward-R2	237.22	315.51	**+33.00%**	386.75	**+63.03%**

Metric	Baseline	RL#7	Gain	RL#8	Gain
ClientRequests-R1	80.80	69.70	-13.74%	111.92	**+38.52%**
Replicated Requests-R1	35.24	61.59	**+74.77%**	30.45	-13.59%
Requests Written-R1	27.57	69.70	**+152.83%**	75.46	**+173.74%**
ClientRequests-R2	178.20	186.21	**+4.49%**	220.57	**+23.77%**
Replicated Requests-R2	27.16	59.94	**+109.69%**	68.20	**+151.15%**
Requests Written-R2	31.86	186.21	**+484.38%**	96.73	**+203.57%**
Avg Reward-R1	143.61	200.99	**+39.95%**	217.84	**+51.69%**
Avg Reward-R2	237.22	429.36	**+81.00%**	385.50	**+62.51%**

Metric	Baseline	RL#9	Gain	RL#10	Gain
ClientRequests-R1	80.80	94.42	**+16.85%**	112.95	**+39.79%**
Replicated Requests-R1	35.24	61.32	**+73.97%**	69.08	**+96.01%**
Requests Written-R1	27.57	94.42	**+242.50%**	112.95	**+309.73%**
ClientRequests-R2	178.20	162.38	-8.88%	205.47	**+15.30%**
Replicated Requests-R2	27.16	91.58	**+237.23%**	98.25	**+261.81%**
Requests Written-R2	31.86	73.29	**+130.01%**	94.26	**+195.82%**
Avg Reward-R1	143.61	250.15	**+74.19%**	294.99	**+105.41%**
Avg Reward-R2	237.22	327.25	**+37.95%**	397.99	**+67.77%**

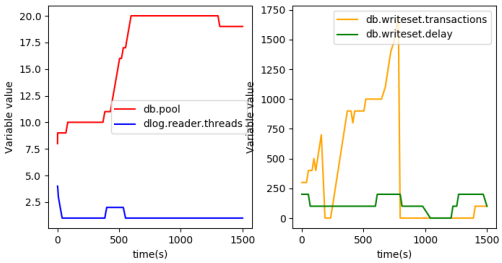
We can also see performance increases in individual metrics of up to 484.38%. By the end of the 10th cycle, improvements were observed in every single metric when compared to the baseline.

Moreover, a deeper look into the results depicted allows for an interesting observation. The performance of a replica (in terms of it's achieved reward) is partly influenced by the performance of the other. The value for the Replicated Requests metric (Table 8) of one replica can only be as high as the value for the Requests Written metric of the other replica. In turn, the Requests Written metric of a replica is limited by the ClientRequests metric of that same replica, since it can only replicate a request after it has executed it. By increasing the performance of one replica's write speed we eliminate a bottleneck that affects the whole cluster, since the other replicas' maximum replication speed is increased. As an example, in the first cycle (RL#1), R1's average write speed to the replication log is of 61.73 *ClientRequests/sec*, which limited R2's read speed of the replication log to 61.75 *ClientRequests/sec*. Note that the monitoring system may, in some situations, miscount a small number of transactions, hence the 0.02 discrepancy. By the last cycle (RL#10) R1's write speed increased to 112.95 *ClientRequests/sec*, allowing for R2's read speed to rise up to 98.25

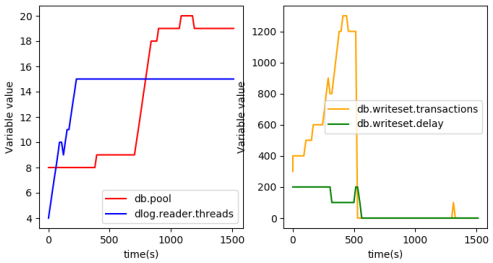
ClientRequests/sec. Not only did R2 dramatically increase it's reading performance, it still has space to improve this speed by a further 24.7 *ClientRequests/sec*.

The actions that were taken in each RL episode are depicted in Figure 24. The figure details the evolution of each considered configuration, on each episode.

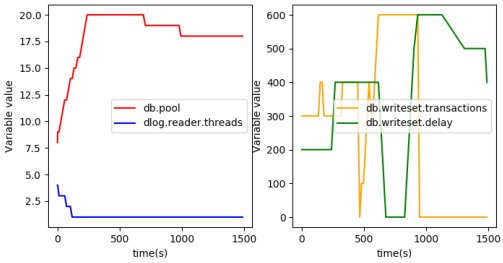
It is possible to observe that the pattern for each configuration variable evolved over the learning cycles. We can also see that it evolved differently for each replica.



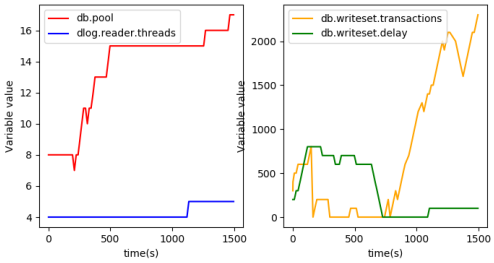
(a) RL 1 - replica 1.



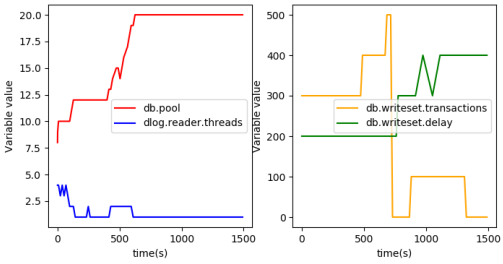
(b) RL 1 - replica 2.



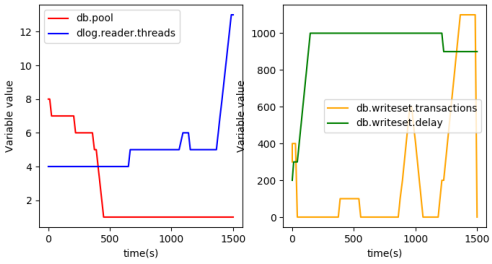
(c) RL 2 - replica 1.



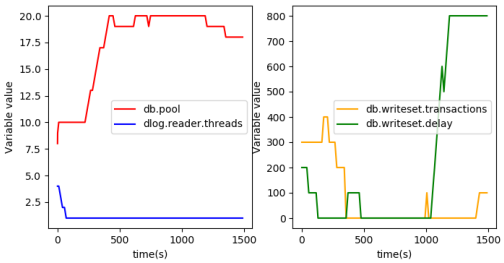
(d) RL 2 - replica 2.



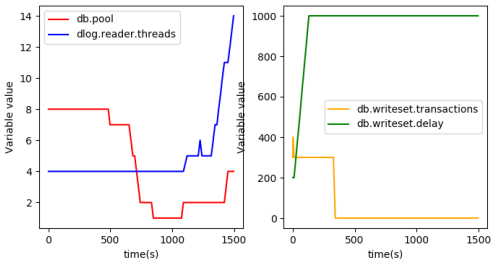
(e) RL 3 - replica 1.



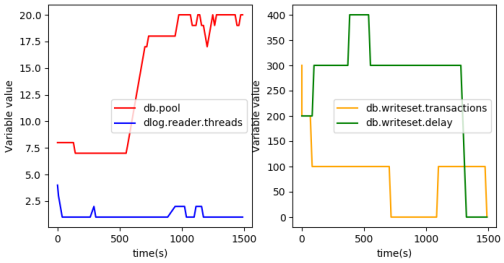
(f) RL 3 - replica 2.



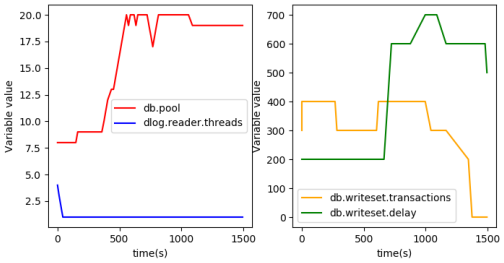
(g) RL 4 - replica 1.



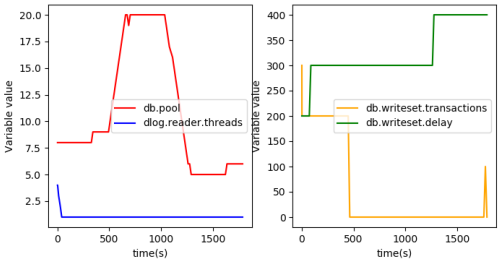
(h) RL 4 - replica 2.



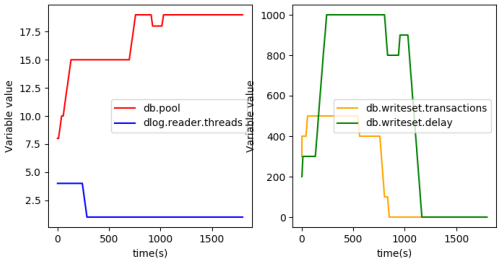
(i) RL 5 - replica 1.



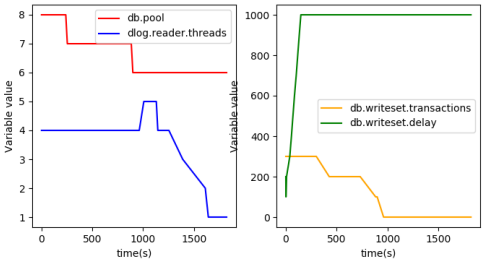
(j) RL 5 - replica 2.



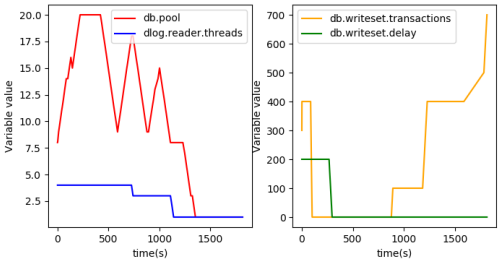
(k) RL 6 - replica 1.



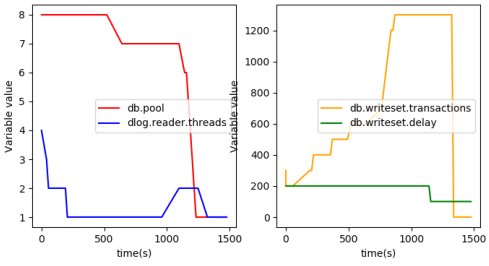
(l) RL 6 - replica 2.



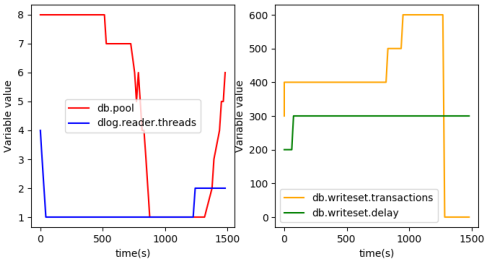
(m) RL 7 - replica 1.



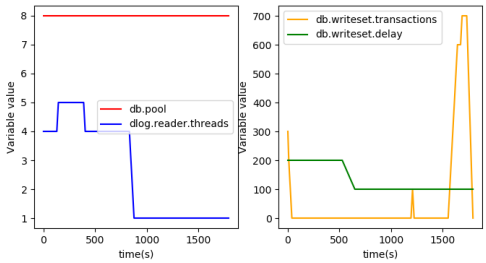
(n) RL 7 - replica 2.



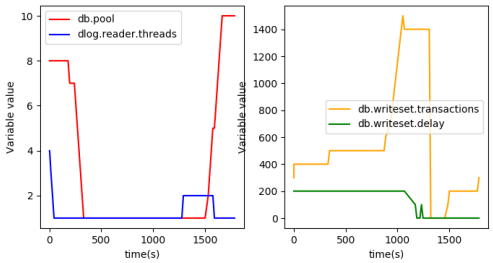
(o) RL 8 - replica 1.



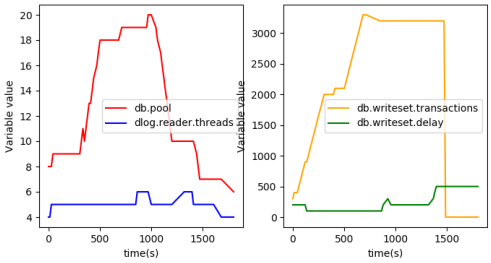
(p) RL 8 - replica 2.



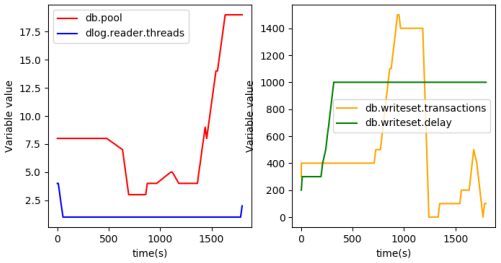
(q) RL 9 - replica 1.



(r) RL 9 - replica 2.



(s) RL 10 - replica 1.



(t) RL 10 - replica 2.

Figure 24: Evolution of actions taken during replica agent evaluation.

Below is an analysis of the adjustment patterns of the replicas for each configuration variable and replica.

db.pool - In replica 1 the database pool is initially increased (or maintained at the initial size) and then decreased by the end of the benchmark run. For replica 2 we see the exact opposite in the last 3 cycles (subfigures 24p, 24r and 24t), the size of the pool is initially decreased and then increased by the end of the run.

dlog.reader.threads - The pattern for the number of threads applying replicated requests is identical for both replicas. The number of threads is decreased so that, most of the time, request replication is carried out sequentially. This behaviour is adopted by replica 1 since episode 1 (24a), but replica 2 only adopts this pattern in the 5th (24j) episode.

db.writeset.transactions - The number of transactions in each write operation tends to follow a pattern where it either is increased or set to 0, in both replicas. The 0 is a special value which removes the limit on a write operation, so that all available client requests will be sent to the replication log. However, when the value of this variable is increased after it has been set to 0, it actually gets set to the lowest possible setting, thus being a mechanism for slowing down the write speed. Sometimes we can observe this behaviour at the end of the benchmark run (e.g., 24g or 24r). Since the RL algorithm works as a black box we can not know why the agent decides to slow down writes to the replication log or even if that is it's intention. It might simply associate both the actions of increasing this variable and setting it to 0 to both be synonymous with an increase in write speed, and so be using them interchangeably, without taking into account this slow down mechanism.

db.writeset.delay - The delay does not impose a perceivable pattern in either replica. Since the RL agent works as a black box when it comes to decision making, there is no way (with the data that is available) to decipher the reasoning behind the actions taken in relation to this variable. It either has

no substantial effect in the performance of the replicas or it has a non-linear behaviour that can't be understood.

Table 9 overviews the results of the final episode, extracted from Table 8. 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 9: Replica agents' final episode 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, the state contains information limited to the replication middleware, holding no variables associated with the underlying hardware, for instance. These characteristics allowed us to train the model in a very short time. However, the low number of state variables probably denies us 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. Also, using so few state variables may be limiting the system's performance, since it might have been able to further improve the performance of the system if it had more elements from which to extract relevant information.

5.2 SHARD AGENT

The shard agent is responsible for managing shard allocations within the entire cluster. So, while in the previous use-case individual replica variables were tuned, the shard agent manages a part of the system that as a global scope. The objective of the shard agent is to dynamically reassign shards between replicas according to changes in the workload and/or to variances in the replicas' hardware usage and performance metric values.

5.2.1 *Setup*

The evaluation was also conducted by running the TPC-C benchmark. This time a more realistic setup was used, where there were two replicas, each running on a different machine, while TPC-C, DistributedLog and the RL agent ran on a third machine. A total of 150 steps were executed, with a minimum time of 10 minutes between each 2 steps. A benchmark was run for 10 minutes on each episode. The episode ended when all of the requests from that benchmark were executed and replicated which, with the number of steps set, equated to 14 episodes. The replicas take more than 2

hours to finish processing all the requests from a 10 minute benchmark. We leave to future work the evaluation of alternative time intervals between steps.

At the beginning of each episode there was an even distribution of 2 shards per replica. Since there were 13 warehouses per log, shard number 4 had two warehouses less than the other shards (13+13+13+11). At the beginning of each episode this shard was always assigned to R2. The initial shard allocation is the same for all learning cycles so that the initial state of the middleware doesn't differ between them. The last 4 episodes were discarded since from around episode 11 the machine running DistributedLog started reaching the limit of RAM memory, which affected the final results for those episodes.

All episodes were adjusted for a faster learning process, with a probability for the action taken being chosen randomly of 20%. This series of cycles was comprised by an average of 11 steps per cycle, updating the Neural Network's weights every 20 steps (less than once per cycle). The gamma value was set to the default value of 0.99 in all episodes, so that future and immediate reward were valued equally.

Separately, a series of 8 benchmarks were run without reallocating any shard. The average reward for these runs was used as a baseline for comparison. During these runs replicas were assigned 2 shards each, same as at the beginning of each learning cycle, which means the load on each replica was approximately the same.

TPC-C was set up in a configuration comprising 50 warehouses with a load of 50 client connections per warehouse. Moreover, the middleware replication component was set up to accommodate 13 warehouses per distributed log instance. The database was reset between each learning cycle. Tests were conducted on a cluster of 3 machines. Machine one ran on Intel(R) Core(TM) i3-2100, 8GB RAM and 250GB HDD, hosting Apache DistributedLog with a 3 bookie setup, TPC-C and the RL agent. The cluster was composed by replicas 1 (R1) and 2 (R2) which ran on identical machines, comprised of an Intel(R) Core(TM) i3-3240, 8GB RAM and 500GB HDD. Machines communicated through a 1Gbit/s switched network.

5.2.2 *Results*

The results of the tests are depicted in table 10. The *Average Reward* values are the average of all the rewards observed during an episode. Measures for replicas R1 and R2 are annotated accordingly. Although the average reward value fluctuated in the first episodes, the final four showed some slight but steady improvement in the average reward, compared to the baseline. In the last learning cycle, an improvement of 21.25% was observed in the average reward relative to the baseline, and all individual metrics were improved, in relation to the baseline. Amongst all the cycles, we saw a maximum increase in average reward of 28.71% and of 69.92% for individual performance metrics.

Table 10: Results of shard agents' evaluation. Baseline and cycle results in client requests per second (Client Requests/sec).

Metric	Baseline	RL#1	Gain	RL#2	Gain
ClientRequests-R1	6.02	4.67	-22.36%	7.68	**+27.62%**
Replicated Requests-R1	3.50	0.74	-78.77%	2.30	-34.30%
Requests Written-R1	5.88	4.67	-20.64%	7.66	**+30.10%**
ClientRequests-R2	4.11	0.75	-81.73%	2.40	-41.50%
Replicated Requests-R2	5.71	4.67	-18.11%	7.67	**+34.28%**
Requests Written-R2	3.98	0.74	-81.31%	2.36	-40.76%
Avg Reward-Cluster	29.64	16.25	-45.16%	30.06	**+1.43%**

Metric	Baseline	RL#3	Gain	RL#4	Gain
ClientRequests-R1	6.02	6.64	**+10.45%**	7.40	**+23.04%**
Replicated Requests-R1	3.50	5.96	**+69.92%**	3.39	-3.26%
Txn Written-R1	5.88	6.58	**+11.74%**	7.40	**+25.78%**
ClientRequests-R2	4.11	6.26	**+52.31%**	3.62	-11.96%
Replicated Requests-R2	5.71	6.59	**+15.42%**	7.38	**+29.32%**
Requests Written-R2	3.98	6.13	**+54.07%**	3.50	-11.96%
Avg Reward-Cluster	29.64	38.16	**+28.72%**	32.70	**+10.31%**

Metric	Baseline	RL#5	Gain	RL#6	Gain
ClientRequests-R1	6.02	6.90	**+14.77%**	4.25	-29.36%
Replicated Requests-R1	3.50	3.64	**+3.88%**	4.67	**+33.38%**
Requests Written-R1	5.88	6.86	**+16.50%**	4.25	-27.79%
ClientRequests-R2	4.11	4.11	**+0.24%**	4.72	**+14.95%**
Replicated Requests-R2	5.71	6.82	**+19.51%**	4.25	-25.66%
Requests Written-R2	3.98	3.81	-4.24%	4.72	**+18.58%**
Avg Reward-Cluster	29.64	32.15	**+8.47%**	26.86	-9.38%

Metric	Baseline	RL#7	Gain	RL#8	Gain
ClientRequests-R1	6.02	7.61	**+26.59%**	6.92	**+14.96%**
Replicated Requests-R1	3.50	3.71	**+5.72%**	4.35	**+23.98%**
Requests Written-R1	5.88	7.29	**+23.92%**	6.76	**+14.94%**
ClientRequests-R2	4.11	4.13	**+0.52%**	4.59	**+11.63%**
Replicated Requests-R2	5.71	6.98	**+22.32%**	6.43	**+12.59%**
Requests Written-R2	3.98	3.83	-3.87%	4.35	**+9.19%**
Avg Reward-Cluster	29.64	33.55	**+13.19%**	33.39	**+12.63%**

Metric	Baseline	RL#9	Gain	RL#10	Gain
ClientRequests-R1	6.02	8.00	**+32.91%**	7.97	**+32.57%**
Replicated Requests-R1	3.50	3.60	**+2.62%**	3.79	**+8.18%**
Requests Written-R1	5.88	8.00	**+35.87%**	7.97	**+35.52%**
ClientRequests-R2	4.11	3.90	-4.97%	4.29	**+4.44%**
Replicated Requests-R2	5.71	7.99	**+39.93%**	7.95	**+39.34%**
Requests Written-R2	3.98	3.76	-5.55%	3.96	-0.67%
Avg Reward-Cluster	29.64	35.24	**+18.89%**	35.94	**+21.25%**

Although the improvement in the cluster's performance is relatively small, these results are still very interesting considering the cluster configuration. As previously described, during the baseline benchmarks, replicas R1 and R2 ran on identical machines and the load on each machine was approximately the same (2 shards per replica). One would expect that if we have a cluster with two identical machines, distributing the load evenly between them would give us the best performance. However, the results show us there were improvements of up to 28.72% in average reward compared to the baseline, when shard reallocation is in effect.

There are some reasons that could justify this difference without implying that the shard agent is responsible for the improvements.

Variation between benchmarks

No two benchmark runs have the same exact average reward, there is always some variation. The point is whether the observed improvement (around 20%) is big enough not to be considered the normal variation. Table 11 shows the reward values for the baseline benchmark runs. It is possible to observe variations between -5% and +6% in relation to the average for those runs, where +20% is a difference approximately 4 times bigger than the one observed during the benchmark runs. It can be concluded (to a reasonable degree of certainty) that the observed improvements where not due to normal variation between benchmark runs.

Table 11: Shard allocation baseline. Reward values in client requests per second (Client Requests/sec).

Run id	Reward-Cluster	Gain
Avg-Cluster	29.64	-
Run 1	29.91	**+0.92%**
Run 2	28.25	-4.68%
Run 3	31.38	**5.85%**
Run 4	30.67	**3.46%**
Run 5	28.52	-3.78%
Run 6	29.55	-0.31%
Run 7	29.69	**+0.17%**
Run 8	29.16	-1.63%

Caching

Caching mechanisms either on the underlying database instances or on the distributed log can also justify variations. Looking at table 11, it can be observed that the last benchmark runs actually had worst results, which renders the conclusion that there isn't any caching process significantly affecting the results. Moreover, database instances are reset between each episode and DistributedLog was reset after the baseline runs.

CPU Operating Temperature

Finally, one could argue that variations in the hardware's surrounding environment could have led to variations in cpu temperature which consequently could lead to thermal throttling during the baseline benchmark runs. Collected hardware monitoring measures state that R1 and R2 ran at a steady 27.8ºC throughout both the baseline and agent evaluation runs. The machine running TPC-C, DistributedLog and the Shard Agent showed temperatures in the 42,0-57.5ºC range approximately, for the baseline runs, with a higher concentration of temperatures in the 42.5-50ºC range. Temperature ranges for the agent runs were identical. Therefore, CPU temperature is also not a root cause for the observed improvement.

Even in an environment where the replicas' underlying hardware was identical, the Shard Agent still managed to obtain a slight but concrete improvement in performance. On a separate test, where the setup and conditions were kept the same but the RL agent and TPC-C were moved to a 4th machine (the one used for 5.1.1), similar results were obtained. The test lasted for 5 episodes and the following reward gain values were observed: -1.02%, $+27.35\%$, $+8.82\%$, $+18.85\%$, $+10.16\%$. The test only lasted for 5 episodes because, once again, memory usage reached it's limit in the machine running DistributedLog.

Let's now take a look at how shards were reallocated during the episodes.

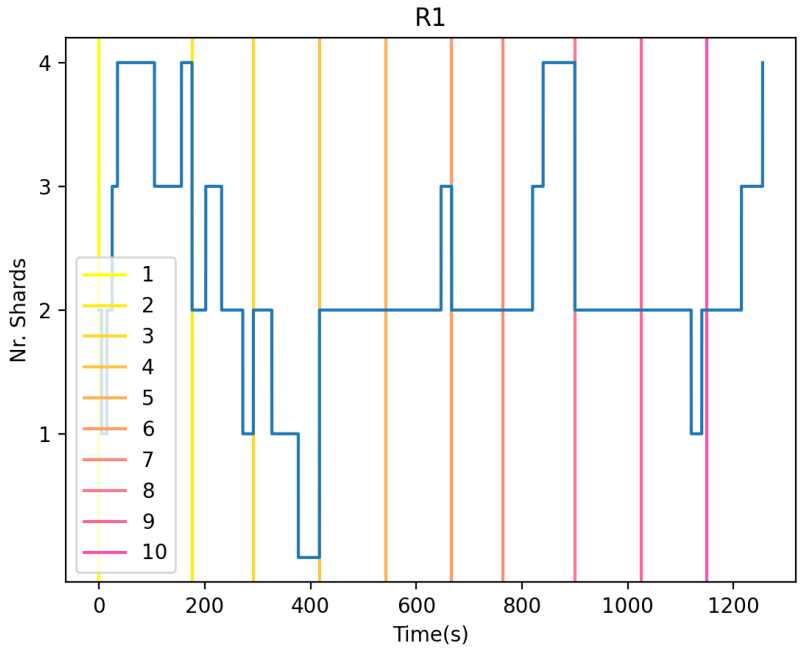


Figure 25: Shard agent evaluation. Shards owned by R1 over time.

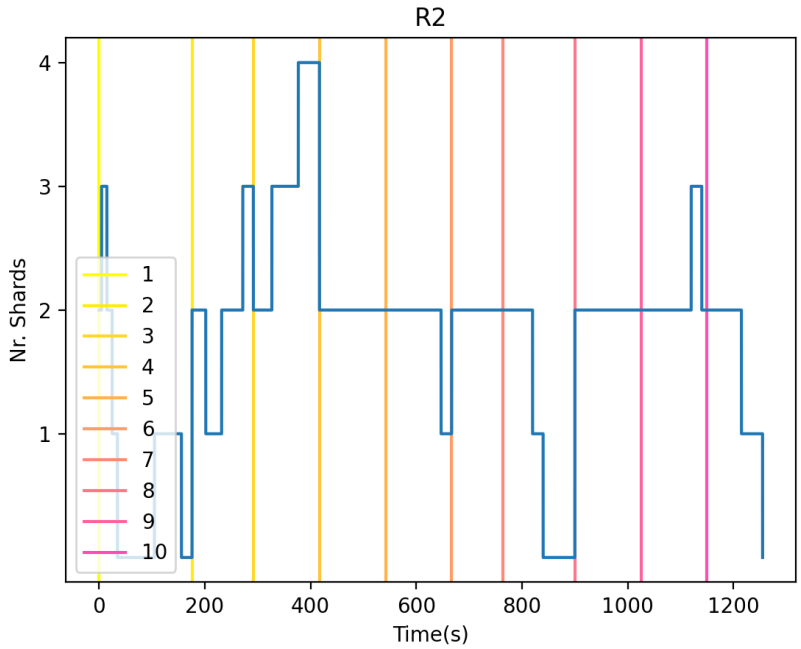
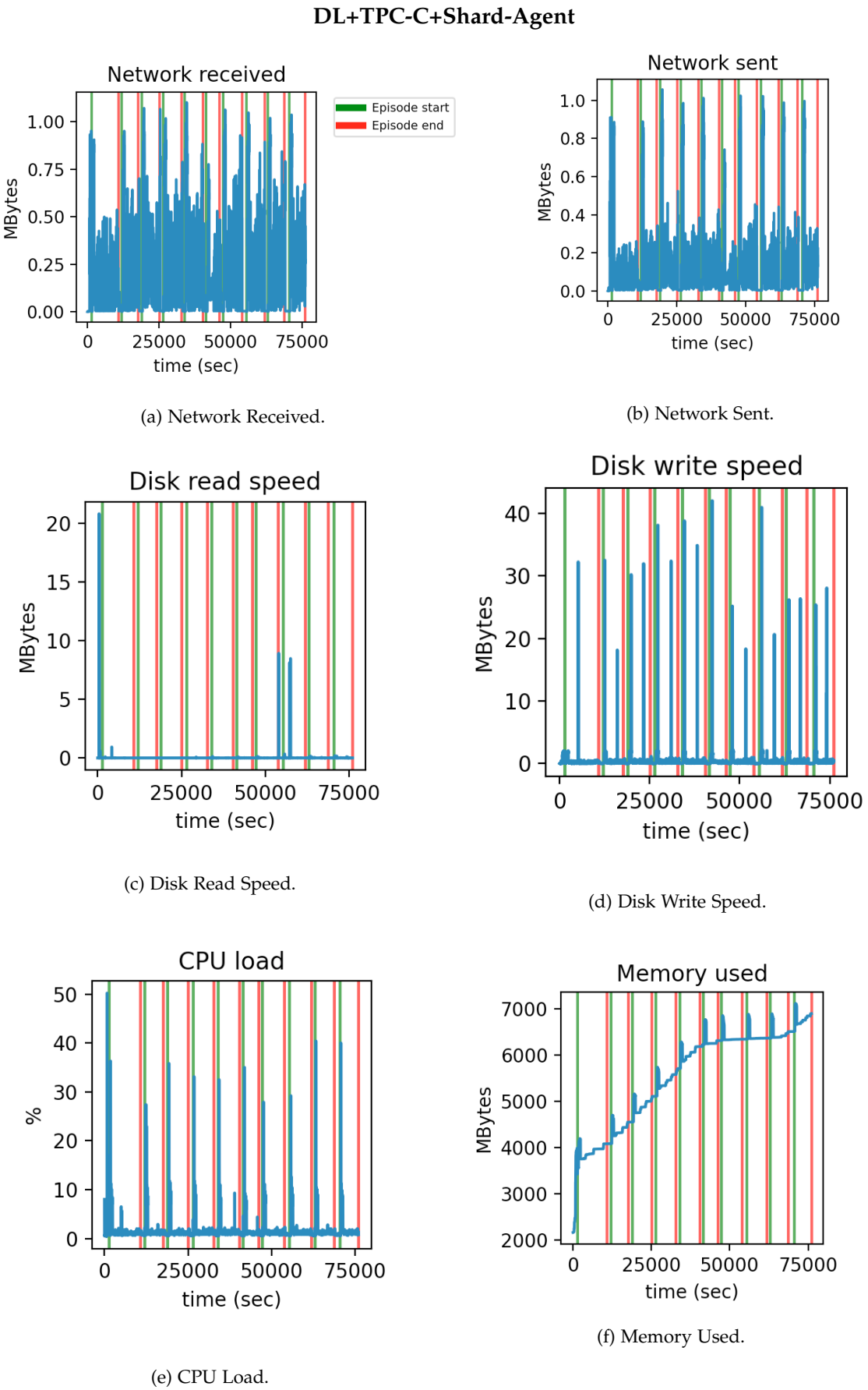


Figure 26: Shard agent evaluation. Shards owned by R2 over time.

Figure 25 and 26 depict, for each replica, how many shards were assigned to it, at a given time, during each episode. Since there are only 2 replicas for a total of 4 shards, the graphs are symmetric

in relation to the $y = 2$ horizontal line. For example, when a replica has 1 shard, the other has 3 and vice-versa. It is possible to observe that the agent has not kept a balanced load between both replicas. It also didn't assign more load on one specific replica when compared with the other, but instead alternated between which replica got more load along the different episodes. The original expectations were that the pattern would stabilize for the latter episodes, keeping 2 shards on each replica most of the time, occasionally deviating from that due to the percentage of actions that are taken at random, but being quickly corrected to the previous allocation pattern, just like what we see between episodes 4 to 6. However results show that this is not true in all cases. For example, there are time intervals where shards are all allocated to the same replica. That's the case with episodes 7 and 10. Since it is not possible to decipher the agents reasoning by looking at the shard allocations, the hardware and software metrics were instead considered.



(a) Network Received.

(b) Network Sent.

(c) Disk Read Speed.

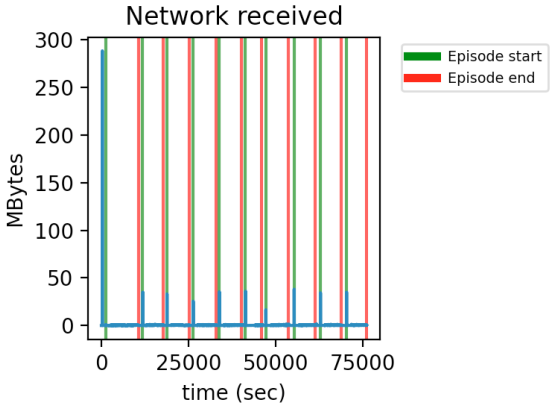
(d) Disk Write Speed.

(e) CPU Load.

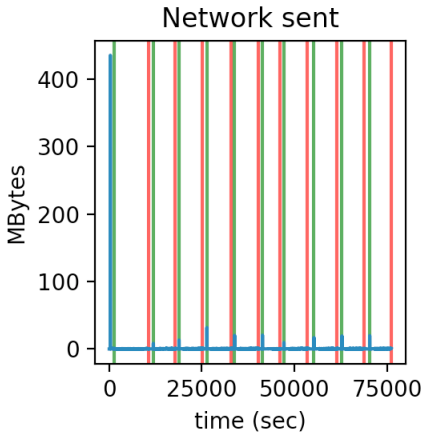
(f) Memory Used.

Figure 27: Shard agent evaluation. Hardware metrics for machine running Distributed Log TPC-C and shard agent with episodes delineated.

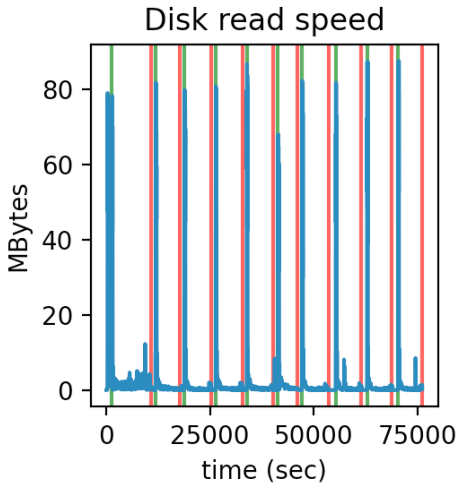
R1



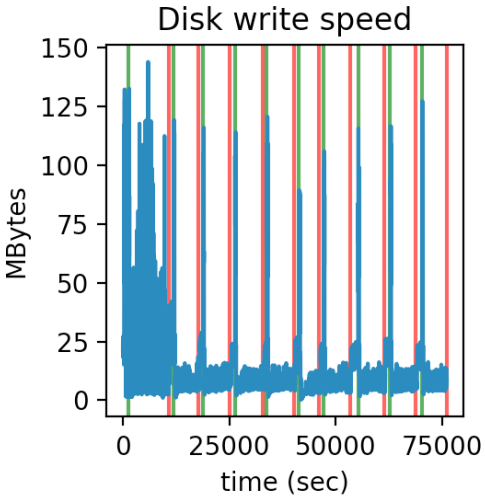
(a) Network Received.



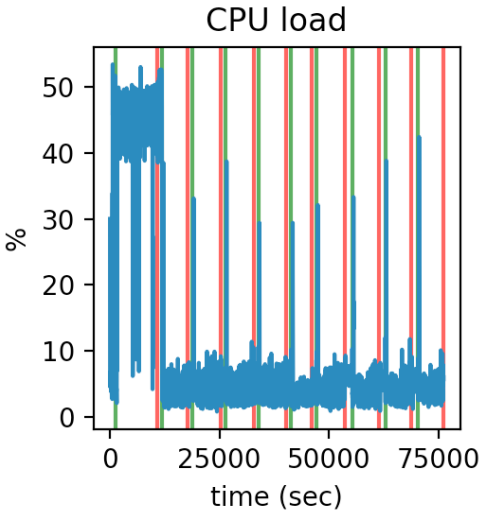
(b) Network Sent.



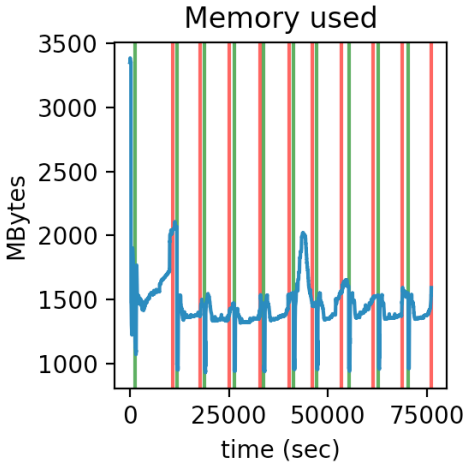
(c) Disk Read Speed.



(d) Disk Write Speed.



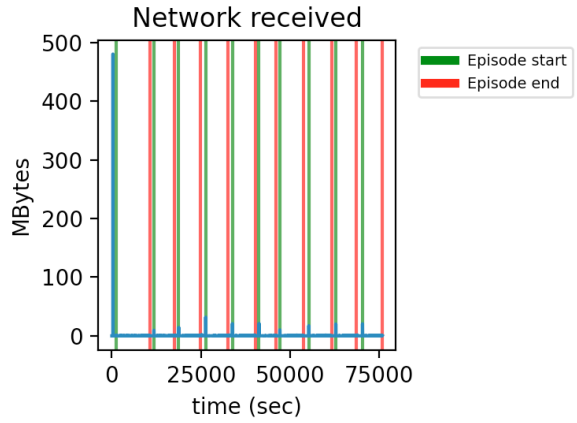
(e) CPU Load.



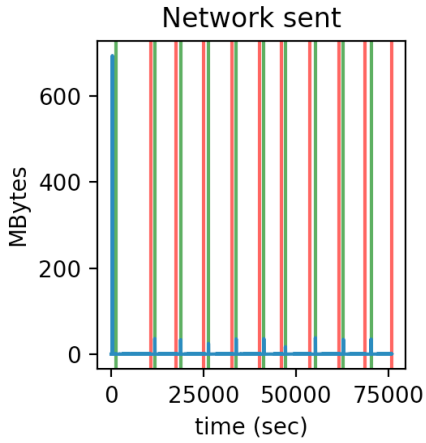
(f) Memory Used.

Figure 28: Shard agent evaluation. Hardware metrics for R1 with episodes delineated.

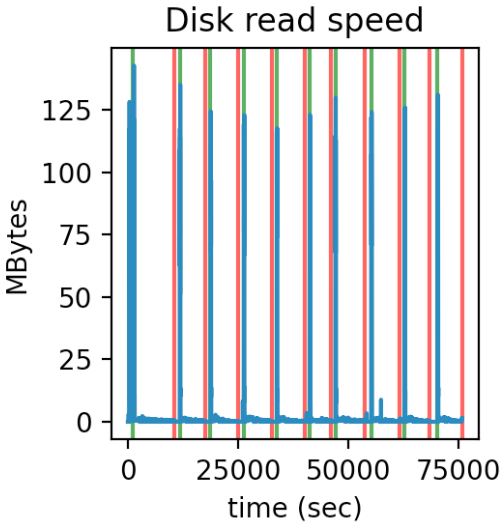
R2



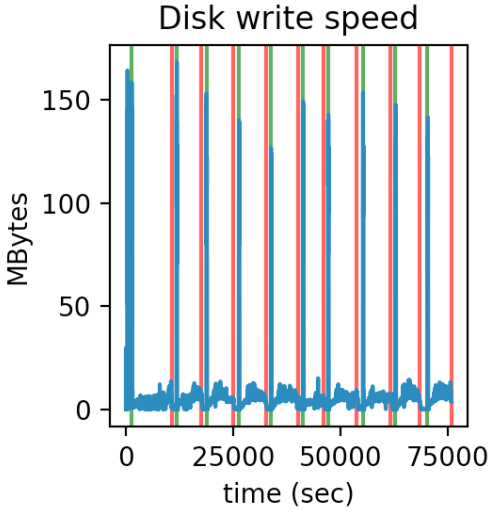
(a) Network Received.



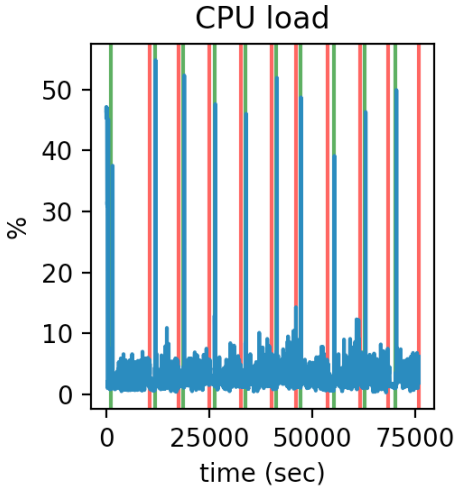
(b) Network Sent.



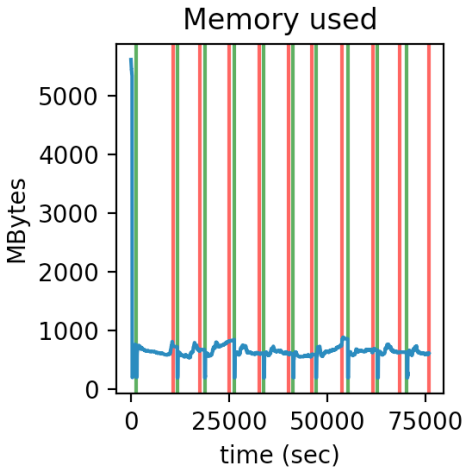
(c) Disk Read Speed.



(d) Disk Write Speed.



(e) CPU Load.



(f) Memory Used.

Figure 29: Shard agent evaluation. Hardware metrics for R2 with episodes delineated.

Looking at Figures 27, 28 and 29 we see similar patterns between them. There is very little network activity overall. CPU load and disk read and write speeds have a spike, on both replicas, at the beginning of each episode, due to the database reset. On R1, CPU load and disk write speed where a lot heavier during the first episode. That would probably have been due to the fact that during episode 1, for most of the time, R1 was processing all 4 shards. Some occasional small spikes in memory usage occur for R1, but in general it stays more or less at a low level, in both replicas, throughout the episodes. The machine running the DistributedLog has a lot more memory usage, which keeps increasing with every episode until eventually it reaches the maximum memory available. At the start of each episode we can see a spike in CPU usage, which is associated to the 10 minute benchmark run. Network usage is overall low. Disk write speed has two spikes, one at the beginning and one at the end of each episode. The first spike is potentially due to the benchmark run. The second one is potentially due to an increase in replication write activity by the end of the episodes, as we'll see later. The high usage of memory suggests that DistributedLog is keeping a lot of information in memory, hence the low disk read levels maintained throughout the episodes.

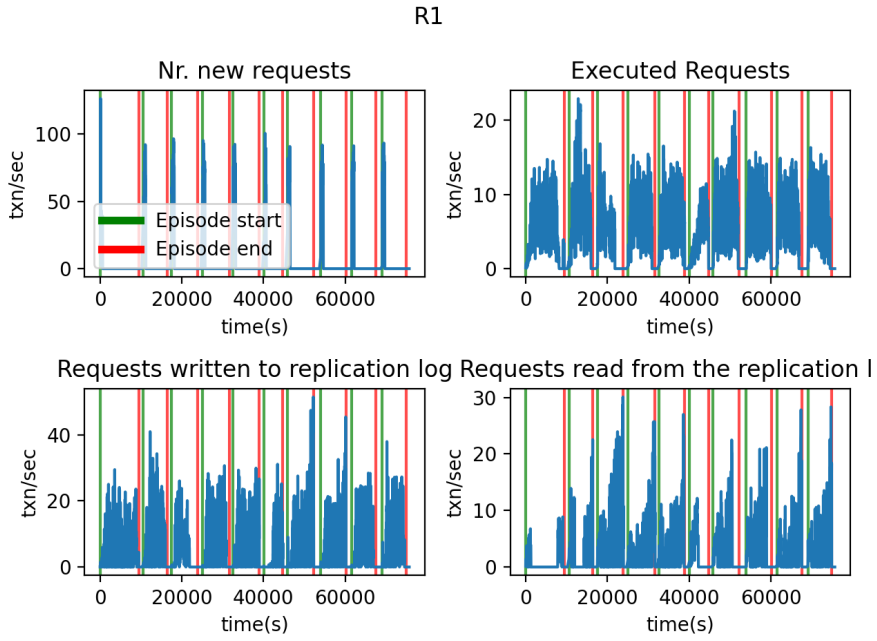


Figure 30: Shard agent evaluation. Evolution of software metrics' values for R1 with episodes delineated.

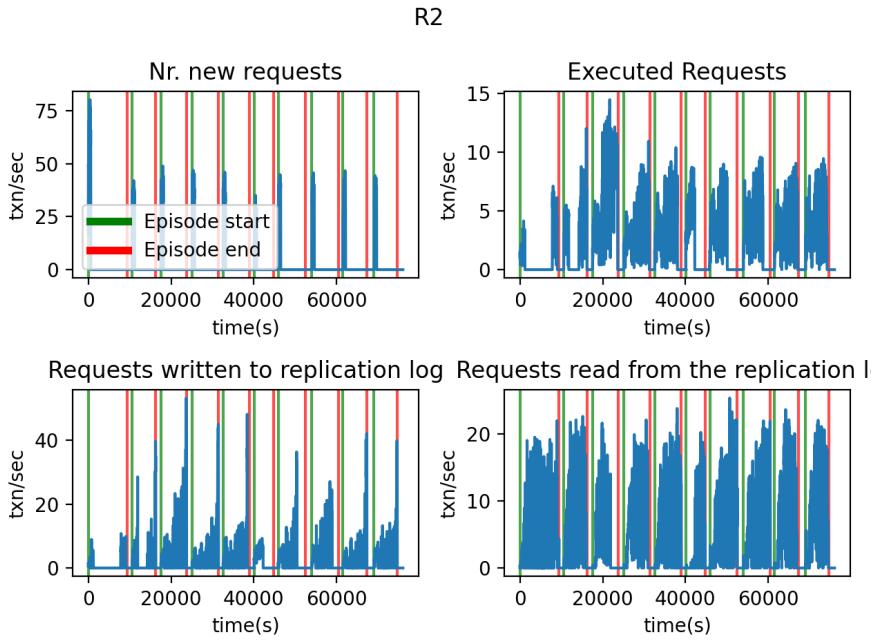


Figure 31: Shard agent evaluation. Evolution of software metrics' values for R2 with episodes delineated.

Figures 30 and 31 show the software metrics side. We can see spikes in new requests at the beginning of each episode. These correspond to new requests being inserted into DistributedLog during the benchmark run (which runs for only 10 minutes). In R2's "Executed Requests" graph we can clearly see, on episode 1, the moment when all shards are attributed to R1. The number of processed requests drops to zero during a significant portion of the time. We can also see the corresponding effect in R2's "Requests written to replication log" graph and in R1's "Requests read from the replication log" graph. From R1's "Requests read from the replication log" and R2's "Requests written to replication log" graphs we might conjecture that as client requests finish being executed, resources are redirected to the replication of requests.

Once again, since the agent's algorithm works as a black box, we can't know what the reasoning behind the shard reallocations is. However, knowing that neither looking at software metrics nor at the hardware monitoring graphics show any identifiable pattern for decision making, and still being able to observe that the agent manages to find a concrete improvement is very surprising. On R2's "Requests read from the replication log" graph, we can see a slight shift to the right. This might suggest that, somehow, the agent is trying to favour the execution of requests during the beginning of the episodes and replication operations at the end, however this shift is not dramatic enough for us to assure with confidence that this is the case.

6

CONCLUSION AND FUTURE WORK

This work highlights how parameter tuning is a complex problem, particularly for distributed systems. It's complexity arises from the large number of variables and the complex relations established between those variables and the performance of the underlying systems. As the complexity of these systems and the need for continuous tuning grows, new opportunities arise for self-tuning capabilities to be embedded into the systems themselves.

Related work shows that the best solutions found are based on ML and more specifically on DRL. It also points out interesting strategies to integrate the use of automatic parameter tuning systems in production environments.

The components that build DRL make it ideal for tackling parameter tuning problems. The use of deep neural networks allows for the modeling of the complex relations the variables establish, even when they are of continuous nature, although they do so in a process that works as a black box. This prevents the clear understanding of how the decisions are made, which might be a downside for systems where explainability is a concern. The use of reinforcement learning, that is, learning from feedback, allows to train a model with no previous knowledge of how the system behaves and enables the introduction of **dynamic** adjustment of parameters. Configuration parameters should not be set as static values, except for systems where the workload is constant. In variable workload environments, such as is the case of production databases, the optimal values for the variables will change, and so, those systems should take dynamic configuration adjustment into account. Previous work shows that DRL is the best solution, so far, for parameter tuning in the scope of individual database instances.

This dissertation details TuneRL, extending the potential of DRL to distributed database systems, and more specifically to the replication middleware component. Results show an increase both in individual replicas' performance when DRL is used to tune replica level variables, as well as in the overall cluster's performance when DRL is used to manage shard reallocation. Most interestingly, DRL shows a tangible improvement in a context where, from intuition, an improvement wasn't expected (see 5.2). It can then be concluded that DRL is a suitable solution for the use cases considered.

The idea of *dynamic parameter adjustment* is not constrained to use in database related systems. Any system which is subjected to variations in workload or even hardware capabilities should, in theory, benefit from continuous adjustment of it's tuning parameters.

The implemented solution, together with the presented middleware, proves to be a very powerful database replication solution, as the previously high throughput capabilities of the middleware are now empowered by DRL optimization. Furthermore, the added system does not force any additional

restrictions on the original middleware, so that it can still be used with any type of database instance that is jdbc compliant.

Many areas of TuneRL can be improved, which we leave to future work. The use of secondary or idle database instances to train the agents, coupled with snapshot or other similar mechanisms, could add a level of protection for production systems as well as minimize resource under utilization. Also, the agents presented here where not evaluated in regards to compatibility across replicas with different hardware characteristics. This means that it is still unclear if an agent trained for a specific replica would also perform well for replicas with different hardware and workload characteristics. Further evaluation should also be conducted with clusters larger than 2 instances to check for scalability of the system. More generally speaking, further tests should be conducted with different underlying database systems, different and/or more state variables (e.g. disk write/read speed) and alternative RL algorithms.

Finally, experiments evaluating the impact of dynamic variable adjustment should be conducted not just in the context of distributed databases, but in the context of other systems, such as publish-subscribe communication services, web servers or even big data processing platforms (e.g. spark [23]).

BIBLIOGRAPHY

[1] Cloudbappliance h2020 project - european cloud in-memory database appliance with predictable performance for critical applications. URL https://cordis.europa.eu/project/id/732051.

[2] Hugo Miguel Ferreira Abreu. High availability architecture for cloud based databases. Master's thesis, Escola de Engenharia, Univesidade do Minho, Braga, Portugal, 2019. Open access from November 2022.

[3] Divyakant Agrawal, Gustavo Alonso, Amr El Abbadi, and Ioana Stanoi. Exploiting atomic broadcast in replicated databases. In *European Conference on Parallel Processing*, pages 496–503. Springer, 1997.

[4] Apache BookKeeper - A scalable, fault-tolerant, and low-latency storage service optimized for real-time workloads, 2018. URL https://bookkeeper.apache.org/.

[5] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, volume 215, 2003.

[6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[7] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. *Information and Computation*, 118(1):158–179, 1995.

[8] Biplob K Debnath, David J Lilja, and Mohamed F Mokbel. Sard: A statistical approach for ranking database tuning parameters. In *2008 IEEE 24th International Conference on Data Engineering Workshop*, pages 11–18. IEEE, 2008.

[9] Dgraph. Not everything can fit in rows and columns. https://dgraph.io. [Online; accessed on the 29th of January, 2020].

[10] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009.

[11] Luis Ferreira, Fábio Coelho, Ana Nunes Alonso, and José Pereira. Towards intra-datacentre high-availability in clouddbappliance. In *CLOSER*, pages 635–641, 2019.

[12] The Apache Software Foundation. Apache tomcat. http://tomcat.apache.org/. [Online; accessed on the 28th of October, 2020].

[13] Pieter Hintjens. *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 2013.

[14] Keras. Keras: Simple. flexible. powerful. https://keras.io/. [Online; accessed on the 16th of October, 2020].

[15] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[16] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. Qtune: a query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment*, 12(12):2118–2130, 2019.

[17] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[18] MySQL. Mysql database service. https://www.mysql.com/. [Online; accessed on the 06th of October, 2020].

[19] Robin L Plackett and J Peter Burman. The design of optimum multifactorial experiments. *Biometrika*, 33(4):305–325, 1946.

[20] PostgreSQL. Postgresql: The world's most advanced open source relational database. https://www.postgresql.org/. [Online; accessed on the 06th of October, 2020].

[21] Heaton Research. The number of hidden layers. https://www.heatonresearch.com/2017/06/01/hidden-layers.html. [Online; accessed on the 17th of October, 2020].

[22] RiskAMP. How latin hypercube compares to standard random sampling. https://www.riskamp.com/latin-hypercube-sampling. [Online; accessed on the 02nd of January, 2020].

[23] Apache Spark. Lightning-fast unified analytics engine. https://spark.apache.org/. [Online; accessed on the 28th of October, 2020].

[24] Stephanie. Latin hypercube sampling: Simple definition. https://www.statisticshowto.datasciencecentral.com/latin-hypercube-sampling/. [Online; accessed on the 2nd of January, 2020].

[25] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[26] TPC-C Benchmark - Standard Specification, February 2010. URL http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[27] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1009–1024. ACM, 2017.

[28] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *arXiv preprint arXiv:1509.06461*, 2015.

[29] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003, 2016.

[30] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, 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*, pages 415–432. ACM, 2019.

[31] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 338–350. ACM, 2017.