UNIVERSITY OF THESSALY
DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

Optimization of the splitting parameters selection on Decision Trees used by dynamically adaptive Markovian State Spaces on elastic resource management under unpredictable loads

MASTER THESIS

Georgios Argyriou

Academic supervisor: Dr. Dimitrios Katsaros
Assistant Professor

Volos, February 2019

Acknowledgements

I would like to thank my supervisors Dimitrios Katsaros and Ioannis Konstantinou for the help they provided. This work is the result of cooperating with the Computing Systems Laboratory of National, Technical University of Athens. Special thanks to Giannis Giannakopoulos for all the help and immediate response whenever I had technical issues with my OpenStack resources.

“Swimming against the current. Lost in a sea of mediocrity”
J.P.

Abstract

Cloud computing has gained popularity over the last decade leading companies and organizations to either offer or use Infrastructure as a Service (IaaS). By utilizing such technologies they succeed in using computer resources according to their everyday needs. In IaaS, the computer resources are shared between many users and usage is much more effective. Even so, when a user binds an amount of resources and deploys an application, the clients may create varying loads, because clients' needs vary during a 24 hour period. Therefore, users of IaaS need a way to increase or decrease their resources according to their clients' needs in order to satisfy the higher loads during peak hours, but not bind more resources than needed during periods of lower traffic. This bind and release process can be achieved either manually, or by following a simple policy that is offered by several administrating systems

In this work we study Tiramola's performance, a system that offers automatic adaptation of the size of a NoSQL database according to user's policy. We use the last version of Tiramola, where the Decision Making module implements Reinforcement Learning algorithms along with adaptive partitioning of the State Space, using Decision Trees. In order to succeed in partitioning the State Space, Tiramola uses metrics as splitting parameters from the cluster of VMs where a NoSQL database (HBase in our case) is deployed and stressed under a load. In the first two phases of experiments we study the behavior of the metrics of the HBase-cluster under linear increasing load and constant load. Based on this data analysis we estimate which one of them can behave better as a splitting parameter when used by Tiramola. In the third phase of experiments, we stress the HBase cluster under sinusoidal load, use the metrics we studied as splitting parameters and verify our evaluation. Tiramola's performance reveals which metrics are efficient as splitting parameters and which are not. In the fourth and last phase of experiments, we configure Tiramola to use the best splitting parameters and study its performance by stressing the HBase cluster under unpredictable load.

Key Words

Distributed Systems, Cloud Computing, Markovian State Space, Resource Management, Elasticity, Data Analysis, NoSQL, HBase, Tiramola

Περίληψη

Η αύξηση της δημοτικότητας των υπολογιστικών νεφών, γνωστός όρος ως cloud computing, έχει οδηγήσει σε προσφορά Υποδομών ως Υπηρεσίες, γνωστές κι ως Infrastructure as a Service (IaaS). Με αυτές οι χρήστες μπορούν να δεσμεύσουν πόρους για εκτέλεση απλών εφαρμογών ή πειραμάτων κατά το δοκούν. Ως αποτέλεσμα, η χρήση των υλικών πόρων διαμοιράζεται ανάμεσα σε πολλούς χρήστες κι αξιοποίηση των πόρων είναι πιο αποτελεσματική. Παρατηρείται πάντα αυξομείωση στη χρήση αυτών των συστημάτων καθώς ο φόρτος που δημιουργούν οι τελικοί χρήστες των εφαρμογών αλλάζει κατά τη διάρκεια της μέρας. Ως εκ τούτου, υπάρχει απαίτηση για αυξομείωση της διαθεσιμότητας των πόρων ώστε από τη μία οι εφαρμογές που τρέχουν πάνω σε τέτοια συστήματα να έχουν συνέπεια και να εξυπηρετούν τη ζήτηση ακόμα και τις ώρες αιχμής, αλλά από την άλλη να μη δεσμεύουν περισσότερους από τους απαιτούμενους πόρους τις υπόλοιπες χρονικές περιόδους. Αυτή η αυξομείωση μπορεί να επιτευχθεί είτε χειροκίνητα, είτε με χρήση απλής τακτικής που προσφέρουν διάφορα συστήματα διαχείρισης υποδομών.

Σε αυτή την εργασία μελετούμε την επίδοση του Tiramola, ενός συστήματος που επιτρέπει την αυτόματη αυξομείωση του μεγέθους μίας NoSQL βάσης δεδομένων ακολουθώντας οποιαδήποτε τακτική ορίσει ο χρήστης. Χρησιμοποιούμε την τελευταία έκδοση του Tiramola, όπου στη μονάδα απόφασης γίνεται προσαρμοστικός διαμοιρασμός χώρου καταστάσεων Μαρκοβιανών μοντέλων. Για να επιτευχθεί ο διαμοιρασμός των καταστάσεων χρησιμοποιούνται οι μετρικές των εικονικών μηχανημάτων όπου τρέχει μια NoSQL κατανεμημένη βάση δεδομένων (HBase) υπό φορτίο. Στις δύο πρώτες φάσεις πειραμάτων εξετάζουμε την συμπεριφορά των μετρικών της συστάδας εικονικών μηχανημάτων της HBase υπό γραμμικά αυξανόμενο φορτίο, αλλά και σταθερό, ώστε να αξιολογήσουμε ποιες από τις μετρικές μπορούν να λειτουργήσουν καλύτερα ως παράμετροι διαχωρισμοί Μαρκοβιανών καταστάσεων από τον Tiramola. Στην τρίτη φάση πειραμάτων επιβεβαιώνουμε την αξιολόγηση των μετρικών/παραμέτρων διαχωρισμού παρακολουθώντας τις επιδόσεις του Tiramola υπό ημιτονοειδές φορτίο προς τη βάση. Στην τέταρτη και τελευταία φάση πειραμάτων, χρησιμοποιούμε τις βέλτιστες παραμέτρους διαχωρισμού και παρακολουθούμε τις επιδόσεις του Tiramola υπό απρόβλεπτο φορτίο.

Λέξεις κλειδιά

Κατανεμημένα Συστήματα, Υπολογιστικό Νέφος, Μαρκοβιανές Καταστάσεις, Ελαστικότητα, Διαχείριση Πόρων, Ανάλυση Δεδομένων, NoSQL, HBase, Tiramola

Contents

List of Figures

List of tables

Chapter 1

Introduction

1.1 Cloud Computing, NoSQL databases and elasticity

The explosive growth of data during the last decade led us to employ new ways of storing and processing them. New kinds infrastructures were created and cloud computing was adopted very quickly both by companies and researchers. The volume of the data is so vast that it became necessary to divide them into more than one machines. Distributed systems were evolved and distributed file systems like the Hadoop DFS were created. Along with distributed storage, platforms of distributed processing were developed like Apache Hadoop, Apache Spark and many more. Furthermore, the computing society needed to find an SQL equivalent for distributed data and NoSQL distributed databases came to play like HBase, Cassandra, Riak, Voldemort and many more.

NoSQL databases are horizontally scalable distributed non-relational storage spaces. They are designed to run on large scale distributed systems, managing the distribution of data and the coordination of machines. Also, they tolerate hardware failures.

One of their most important characteristics of NoSQL databases is elasticity. It makes them the most suitable for using Infrastructure as a Service (IaaS) offered by cloud computing platforms. IaaS gives the ability of elastically scale up or down according to the user's needs. Elasticity is a very important ability giving the opportunity to users to adapt to the incoming traffic caused by clients. Adapting to the resources according to incoming traffic can reduce the cost during periods of low incoming load, while keeping an application available during the periods of high demand.

Tiramola is a system that allows automating elasticity of NoSQL databases according to a user defined policy. Unlike other systems, Tiramola is not using simplified methods to automate elasticity like defining a simple threshold or asking from the user to set the conditions. Tiramola's last version is using Reinforcement Learning algorithms such as Markov Decision Processes and Q-Learning enriched with adaptive State Spaces by utilizing Decision Trees. This enrichment allows Tiramola to better adapt the State Space and capture the complexity of the system.

1.2 Thesis subject

In this work, we are going to use the last version of Tiramola and evaluate its behavior when unpredictable load runs against an HBase cluster. Tiramola's last version has already been evaluated as an improvement in comparison to older versions that implement more traditional Reinforcement Learning algorithms, but not yet been tested adequately as to which are the optimal splitting parameters. Also, all the related works about automating elasticity with Tiramola were always using sinusoidal load, but in this work we are going to experiment using unpredictable loads.

The first challenge hides a bigger topic that has never been fully covered in any previous relevant work, the HBase cluster's behavior. Every Tiramola version has a Decision Making module that implements Reinforcement Learning algorithms. That's how Tiramola decides about the size of the cluster. For doing so, every Tiramola version uses the metrics of a NoSQL cluster and these metrics have never been analyzed adequately. So, in the first two parts of the experiments we study the behavior of the HBase cluster while it is stressed. We monitor the metrics of the cluster and considering how the splitting parameters in Decision Trees are used, we evaluate the suitability of the metrics as splitting parameters. During the third phase of experiments we verify our evaluation and finally decide which of the metrics can be the optimal splitting parameters for Tiramola's Decision Trees. That's how we meet the challenge about using optimally the last version of Tiramola.

The second challenge is split into two smaller ones. In the first phase we try to find the level of randomness of an unpredictable load. When we find it, in the second phase we configure Tiramola's range of actions in order to have a fair encounter between the big changes of the load and Tiramola's flexibility. Then, we study and evaluate Tiramola's behavior against the most reasonably unpredictable loads.

1.3 How this work is organized

In the second chapter we present all the tools and platforms that are used in this work and give an overview of our infrastructure.
In the third chapter we dive into Tiramola and explain its workflow in a technical way and give an overview of what happens when we use Tiramola.
In the fourth chapter we present the first three round of experiments. By analyzing the HBase cluster's metrics we accomplish to describe how HBase reacts when stressed. Also, by knowing how the splitting algorithm works, we can make solid assumptions about which one of them can be used efficiently as a splitting parameter. In the third round, we verify which parameters are more efficient

In the fifth chapter we are testing Tiramola by using the optimal splitting parameters. We run several types of unpredictable loads against the HBase, study its behavior, extend its flexibility and finally evaluate it.
In the sixth and last chapter, we present our conclusions.

Chapter 2

Technical Aspects

2.1 Technical Overview

In order to test automation of the elasticity of a NoSQL database we need to reproduce the whole environment. So, except of the cluster of machines that have a NoSQL database installed, we also want machines that will act as clients of the database and make the queries, a system that can monitor everything and of course, we want the tool that decides automatically to expand or contract the NoSQL cluster.

We use OpenStack to create all the virtual machines (VMs) we need. By using OpenStack, we create cluster of VMs where a NoSQL database is installed, HBase in our case, and a second cluster of VMs that will act as clients. In each client VM we install the YCSB tool which can insert records into HBase, make either read or update queries and even delete the records. For monitoring everything, we use Ganglia. The Open Stack installation offered by the Computer Science Laboratory of the National and Technical University already has Ganglia that monitors the installation (outside of our VMs). We also install another Ganglia on the HBase cluster, because we want metrics from inside of the HBase cluster. Last but not least, Tiramola is installed on the master of the HBase cluster and is in charge of everything: start/stop the clients, define what kind of load the clients create, get all kinds of metrics, decide the size of the cluster and send all commands either to the HBase cluster about start/stop/restart or to the OpenStack installation about add/remove VMs to the HBase cluster.

2.2 OpenStack

OpenStack [7] is a cloud operating system that controls large pools of compute, storage and networking resources throughout a datacenter, all managed through a dashboard that gives administrators control while empowering their users to provision resources through a web interface.

OpenStack began in 2010 as a joint project of Rackspace Hosting and NASA. As of 2016, it is managed by the OpenStack Foundation, a non-profit corporate entity established in September 2012 to promote OpenStack software and its community. OpenStack is a free and open-source software platform for cloud computing, mostly

deployed as infrastructure-as-a-service (IaaS), whereby virtual servers and other resources are made available to customers. The software platform consists of interrelated components that control diverse, multi-vendor hardware pools of processing, storage, and networking resources throughout a data center. Users either manage it through a web-based dashboard, through command-line tools, or through RESTful web services.

2.2.1 OpenStack components

OpenStack Compute, also known as ***Nova***, is a platform whose aim is to manage the OpenStack infrastructure. It provides an interface and an API that allows the management of large networks of virtual machines and scalable architectures. It is written in Python and is designed to scale horizontally on standard hardware with no proprietary requirements.

Imaging Service manages the storage of the images of virtual machines that can later be used as a template for new ones. It provides a RESTful API to perform queries for information about the images hosted on different storage systems.

Object Storage is a storage space that is designed for long term storage of large volumes, and can host up to multiple petabytes of data. Objects and files are written to multiple disk drives spread throughout servers in the data center, while data replication is used to provide data integrity across the cluster.

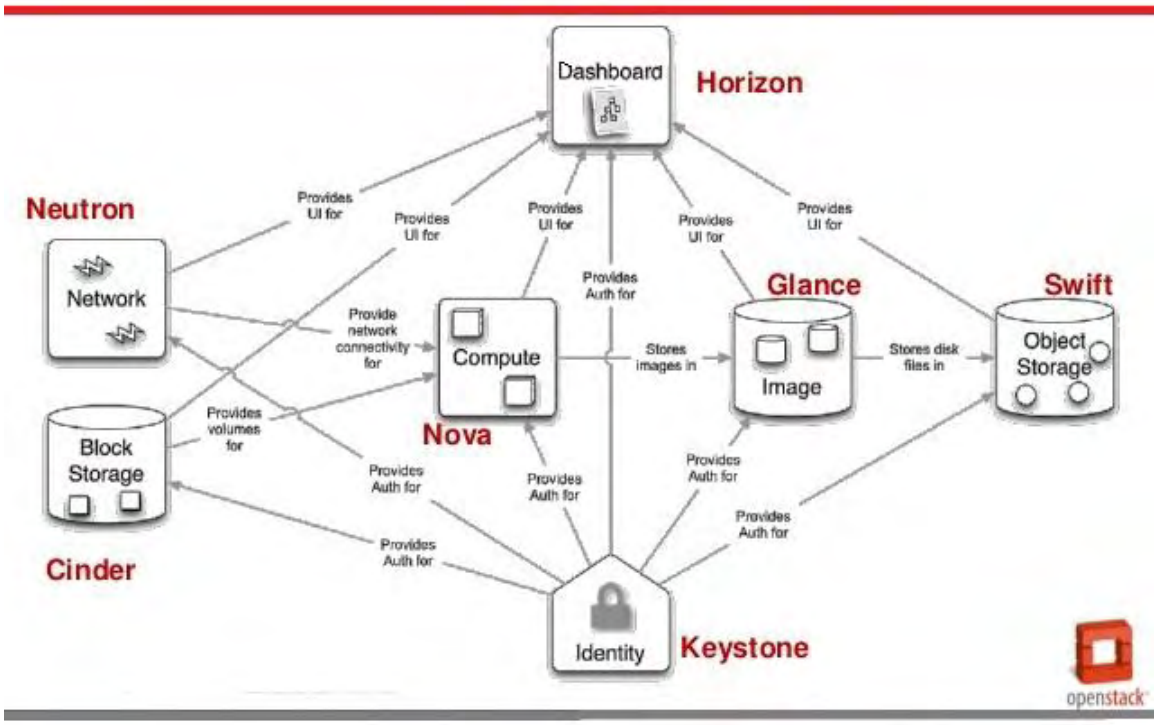

Figure 2.1: *OpenStack components*

The OpenStack services can be accessed through the **OpenStack Dashboard**, ***Horizon***, which provides a graphical interface for users and administrators to access, provision, and automate cloud-based resources. Its design also accommodates third party products and services, such as billing, monitoring, and additional management tools.

OpenStack Identity, ***Keystone***, provides a mapping of users to the OpenStack services they can access. It acts as a common authentication system across the cloud operating system, and supports multiple forms of authentication including standard username and password credentials, token-based systems and AWS-style logins

2.3 Hadoop

Installing Hadoop is a prerequisite for installing HBase. Hadoop [9] is consisted of two main parts, the Hadoop Distributed File System, known as HDFS and the MapReduce which is a programming model for data processing. HBase is using HDFS for storing its data. The HDFS is a distributed file system designed to run on commodity hardware. It is an open source implementation of the ***Google File System*** (GFS) [10] and is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware providing scalability and fault tolerance.

2.3.1 HDFS architecture

HDFS [8] uses a master/slave architecture. The ***Namenode*** takes on the role of the master, and is responsible for coordinating the filesystem and providing access to its files to the clients. Even though data in the HDFS are stored in multiple physical machines, the Namenode maintains a traditional hierarchical file organization. Clients can create files and directories, move and rename them in a manner similar to other existing file systems. Any change to the file system is recorded by the Namenode, which is responsible for maintaining the file system namespace. If the Namenode is not active, clients lose the ability to access the data stored in the HDFS, making it the single point of failure of the system. However, in order to increase reliability, a secondary Namenode is active at all times, and can recover the file system in case of a Master failure.

The slaves in HDFS are called ***Datanodes***, and their responsibility is to store file data and serve read and write requests from the file system's clients. At the same time, they perform block creation, deletion and replication upon instruction of the Namenode. Each file in the file system is stored in multiple equally sized ***blocks*** (typically 64MB), and each of these blocks is hosted in multiple Datanodes in order to increase fault

tolerance. It is possible for applications to specify or change the ***replication factor*** for each separate file.

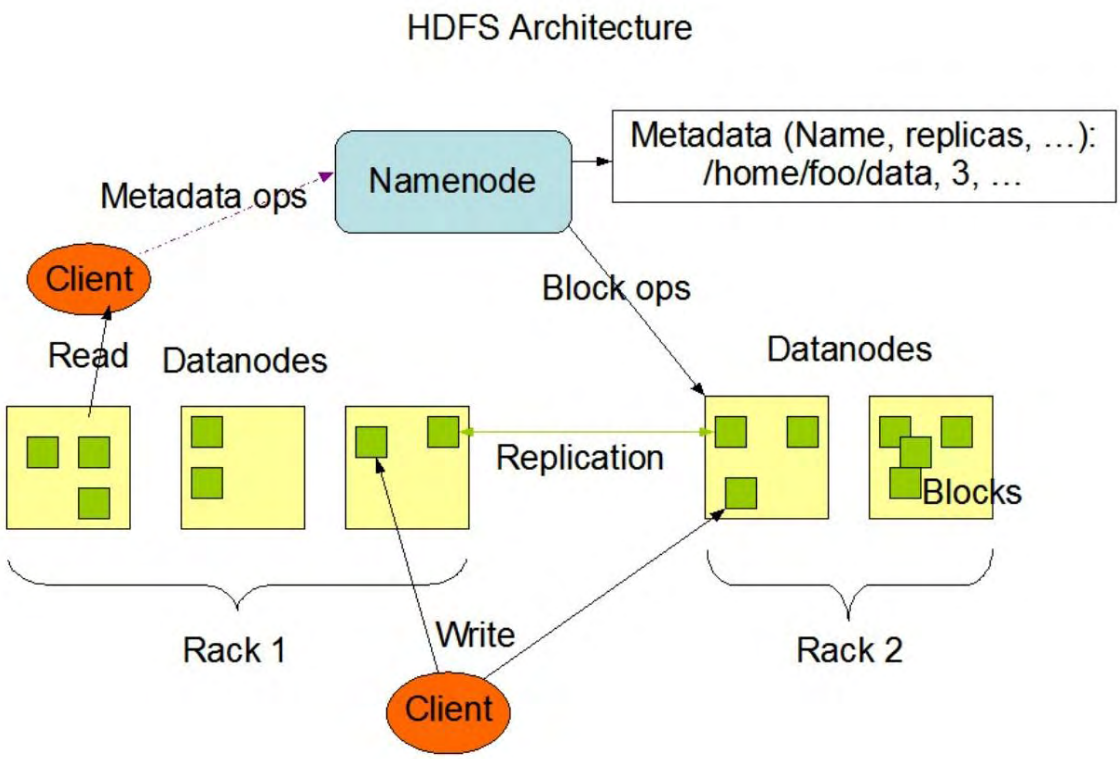

Figure 2.2: *HDFS architecture*

In order for the Namenode to have an up-to-date knowledge of the active blocks in the system, ***Heartbeat*** messages are periodically sent to it from each of the Datanodes. If a Datanode fails to transmit a heartbeat message, the Namenode assumes that the Datanode is dead, stops forwarding new requests to it and attempts to quickly restore the replication factor of its blocks.

The placement of the blocks is decided by the Namenode. The criteria by which this is done is not only to increase fault tolerance, but also to improve performance. In the common case where the replication factor is three, HDFS's placement policy is to put one replica on one node in the local rack, another on a node in a different (remote) rack, and the last on a different node in the same remote rack. This policy reduces the required communication between different racks during writes, while at the same time does not leave the system vulnerable to a single rack failure. However, it does reduce the aggregate network bandwidth used when reading data since a block is placed in only two unique racks rather than three.

2.4 HBase

HBase [12] is an open source, distributed database for storing structured data. Its design is based on Google's ***BigTable*** [13], and runs on top of the HDFS to enhance its storing capabilities. Its data model is different from traditional relational databases. It does not support a structured query language like SQL, but instead uses a key/value model where data are organized in columns.

2.4.1 HBase building blocks

Table. The biggest building block in the database.
Row. Each table consists of a number of rows. Each row possesses a unique key through which it can be identified, and all rows within a table are sorted based on that key. This enables the programmer to control the way data are stored and allows for easy and efficient access to ranges of rows.
Column Family. Data within each row are split to separate column families that are the same for each row and need to be specified upon table creation (even though some rows may not contain data in all column families). Data stored within each column family are also physically stored in adjacent locations in order to more efficiently serve queries requesting data from them.
Column. Each column family contains a number of columns. Unlike column families, columns are allowed to differ from row to row, and can change dynamically.
Cell. A combination of a row key, a column family and a column uniquely identifies a cell. Each cell stores a byte array, which is its value.
Timestamp. HBase has a built-in data versioning and recovery mechanism through the use of its *timestamps*. Instead of storing a single value in each cell, HBase stores a number of recent values. That number can be configured to be different for each column family, and is by default equal to three. If not specified, HBase will store data using the current timestamp and read the data with the latest timestamp, though the user is free to read and write the versions of the data she specifies

2.4.2 HBase architecture

HBase follows a master-slave architecture [11] consisted of the following components:
Master Server. The Master Server in HBase holds the metadata for all the tables stored in the database, and performs schema changes and table creation or deletion operations.

At the same time, it controls the distribution of the regions among the Region Servers in order to evenly balance the workload.

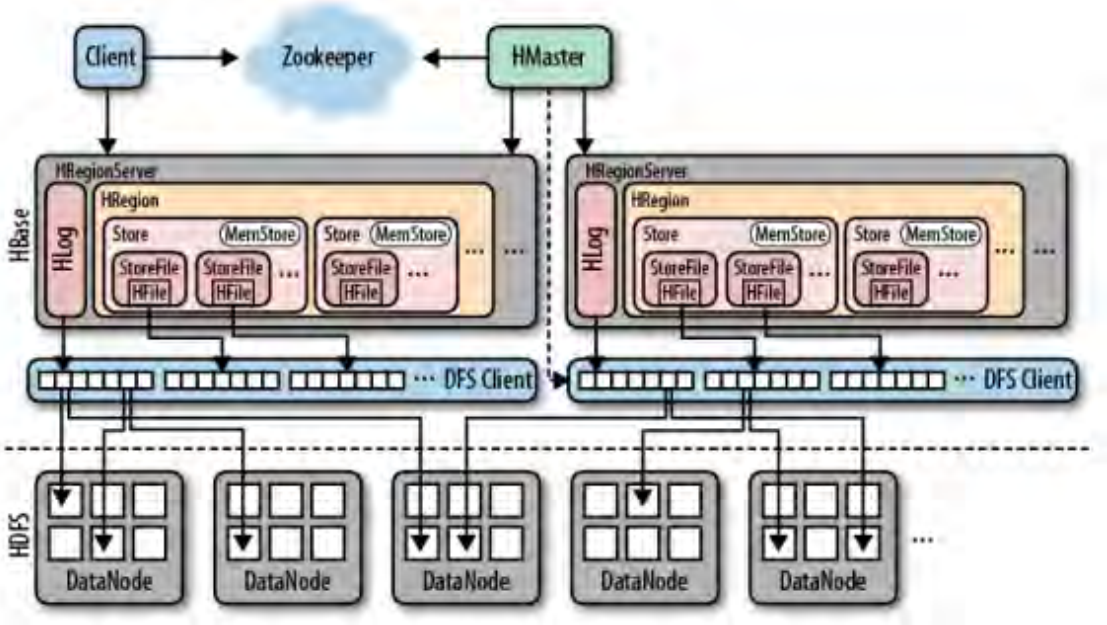

Figure 2.3: *The HBase architecture*

Region Servers. Each Region Server is responsible for serving and managing a number of regions. Even though data stored in the HDFS are spread across different physical locations, each region server stores the data that correspond to the regions it serves within the local HDFS DataNode in order to be able to serve requests locally.

ZooKeeper. It is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. HBase uses ZooKeeper to track the state of the servers in the cluster and handle communication between the master and the region servers.

HBase's architecture allows it to easily scale and store large amounts of sparse data. The fact that it runs on top of HDFS provides high availability and fault tolerance, and makes HBase easy to integrate with other tools within the Hadoop ecosystem, such as MapReduce. Finally, having only a single server responsible for each piece of data, allows it to guarantee strong consistency and perform atomic row operations.

2.5 Yahoo Cloud Serving Benchmark

The Yahoo! Cloud Serving Benchmark (YCSB) [14] is an open-source specification and program suite for evaluating retrieval and maintenance capabilities of

computer programs. It is often used to compare relative performance of NoSQL database management systems.

The YCSB Client is a Java program for generating the data to be loaded to the database, and generating the operations which make up the workload. The architecture of the client is shown in figure 2.4. The basic operation is that the workload executor drives multiple client threads. Each thread executes a sequential series of operations by making calls to the database interface layer, both to load the database (the load phase) and to execute the workload (the transaction phase). The threads throttle the rate at which they generate requests, so that we may directly control the offered load against the database. The threads also measure the latency and achieved throughput of their operations, and report these measurements to the statistics module. At the end of the experiment, the statistics module aggregates the measurements and reports average, 95th and 99th percentile latencies, and either a histogram or time series of the latencies.

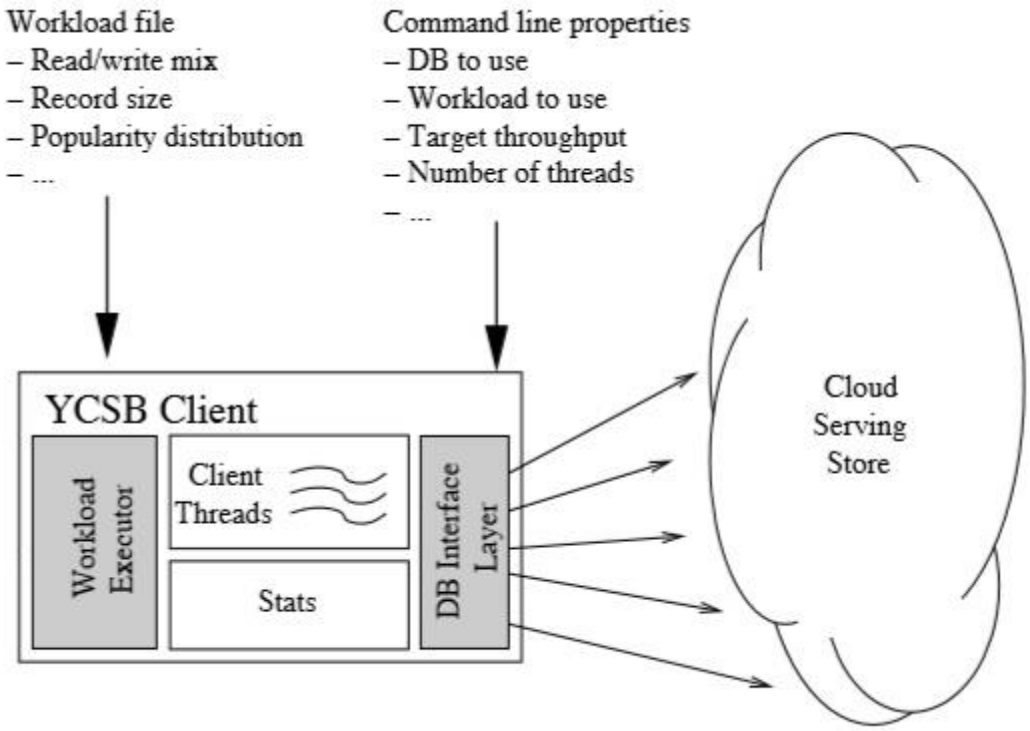

Figure 2.4: *The YCSB client architecture*

The client takes a series of properties (name/value pairs) which define its operation. By convention, we divide these properties into two groups:

Workload properties. Properties defining the workload, independent of a given database or experimental run. For example, the read/write mix of the database, the distribution to use (zipfian, latest, etc.), and the size and number of fields in a record.

Runtime properties. Properties specific to a given experiment. For example, the database interface layer to use (e.g., Cassandra, HBase, etc.), properties used to initialize that layer (such as the database service hostnames), the number of client threads, etc.
Thus, there can be workload property files which remain static and are used to benchmark a variety of databases. In contrast, runtime properties, while also potentially stored in property files, will vary from experiment to experiment, as the database, target throughput, etc., change.

2.6 Ganglia

Ganglia [16] is a scalable distributed monitoring system for high performance computing systems such as clusters and grids, developed by the University of California, Berkeley. It is based on a multicast, listen/announce protocol to monitor the state of the cluster, and uses a tree of point to point connections between representative cluster nodes to federate clusters and aggregate their state. Data are represented in XML format, exchanged using the XDR protocol and stored and visualized with the RRD tool. It manages to achieve very low per node overhead and high concurrency, and is available in a wide range of operating systems.

2.6.1 Ganglia architecture

The Ganglia's components [15] are:

gmond. The Ganglia Monitoring Daemon is installed in every node of the cluster from which metrics are to be collected. Its job is to collect the required metrics with the help of the operating system, as well as announce them to a multicast channel through UDP. It is organized as a collection of threads, most of which are assigned with the task of collecting data for a specific metric. The *collect and publish* thread takes on the responsibility of gathering the metrics collected by the local threads and publishing it on a well-known multicast channel in periodic messages called *heartbeats*. The *listening threads* are responsible for listening on the multicast channel for data transmitted by other nodes and storing it in a local hash table. This allows the data for the whole cluster to be available through any one of its nodes. Finally, a number of *XML export threads* accept and process client requests to provide access to that data

gmetad. Federation in Ganglia is achieved using a tree of point-to-point connections amongst representative cluster nodes to aggregate the state of multiple clusters. At each node in the tree, a Ganglia Meta Daemon periodically polls a collection of child data sources, parses the collected XML, saves all numeric, volatile metrics to

round-robin databases and exports the aggregated XML over TCP sockets to clients. Data sources may be either *gmond* daemons, representing specific clusters, or other *gmetad* daemons, representing sets of clusters. Data collection in *gmetad* is done by periodically polling a collection of child data sources which are specified in a configuration file, dedicating a unique data collection thread to each child source. Collected data is parsed in an efficient manner to reduce CPU overhead and the memory footprint.

RRDtool Storage and visualization of the historical monitoring information for the grid is managed by *Round Robin Database*. *RRDtool* is specialized in storing time series data and is able to maintain different time granularities ranging from minutes to years in compact, constant size databases. Additionally, *RRDtool* is able to plot the historical trends of these metrics on graphs that are used by the *Ganglia PHP web front-end*, to be presented through a web interface

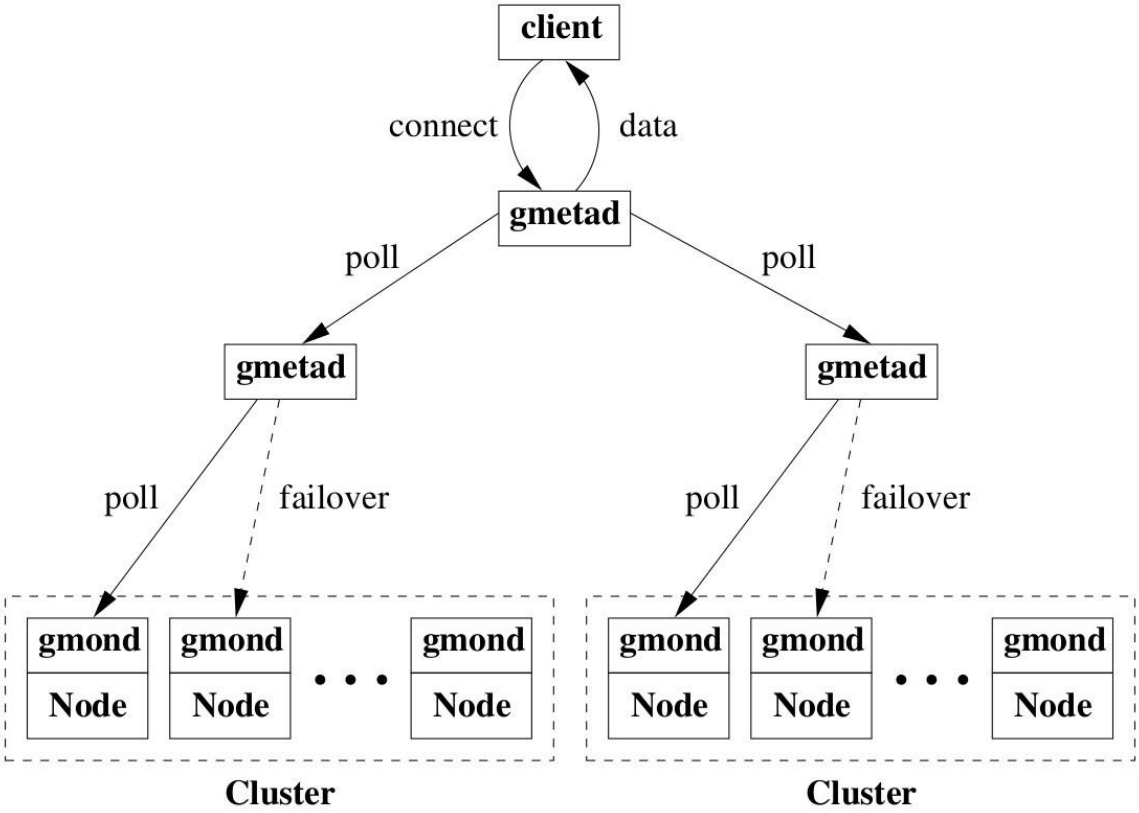

Figure 2.5: *The Ganglia architecture*

2.7 Tiramola

Tiramola [3] is a modular cloud-enabled framework for monitoring and adaptively resizing NoSQL clusters. Its implementation is open-source, and contains modules that can control a number of different NoSQL databases, including Cassandra, HBase, Riak and Voldemort.

2.7.1 Tiramola Architecture

Tiramola [4] is an open-source project that delivers automatic resource allocation for NoSQL clusters. It features a modular architecture illustrated in figure. 2.6. The Decision Making module incorporates both the user-policy defined through an optimization function as well as cluster-side and client-side monitored metrics and periodically decides on cluster resize actions. It outputs resize action to the Cloud Management module that interacts with the cloud vendor in order to release or acquire more virtual machines. The Cluster Coordinator is then responsible for orchestrating the addition and removal commands relative to the particular NoSQL cluster in hand. The Monitoring module maintains up-to-date performance metrics collected from both cluster nodes and client nodes.

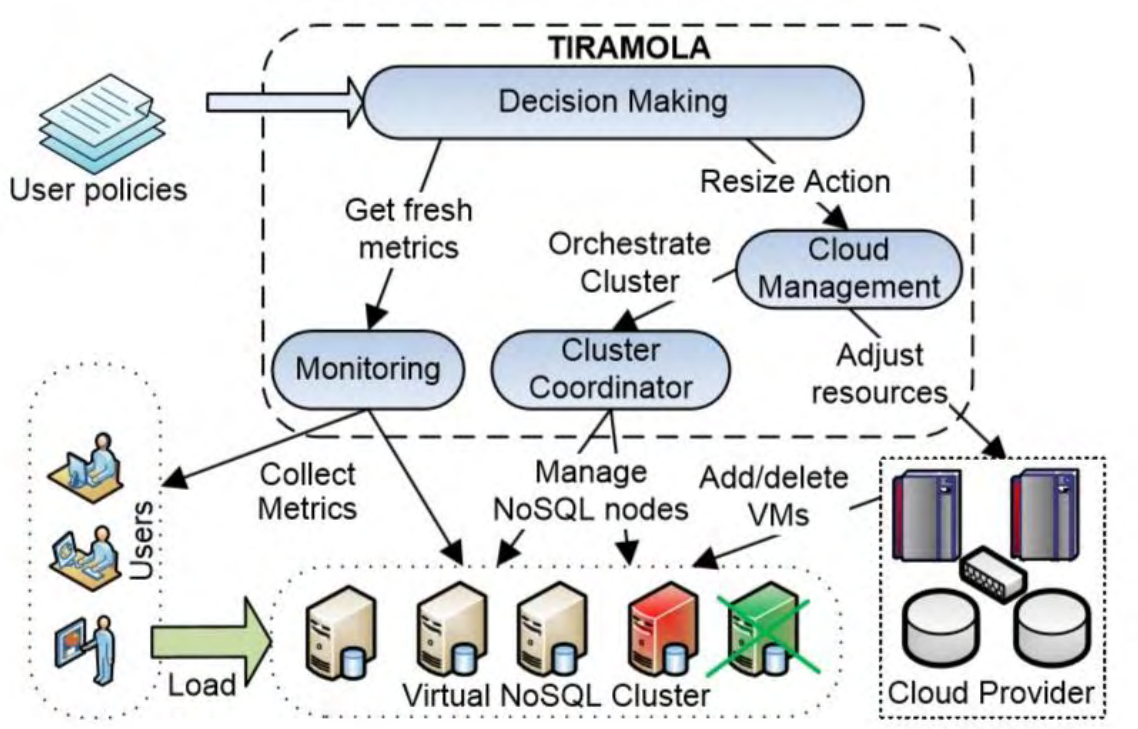

Figure 2.6: *The Tiramola architecture*

Decision Making Module. This module is responsible for deciding the appropriate cluster resize action according to the applied load, cluster and user-perceived performance and optimization policy. Tiramola formulates this process as a Markov Decision Process (MDP) that continuously identifies the most beneficial action relative to the current system state. The goals are defined through a reward function that translates the optimization each application wishes to adhere to. Upon reaching a resize decision, the module forwards this command to the Cloud Management module.

Monitoring. Tiramola uses Ganglia, a scalable distributed monitoring tool that allows remote collection of live or historical cluster statistics (such as CPU load averages, network, memory or disk utilization, number of open client threads, etc) through its XML API.

Cloud management. The system interacts with the cloud vendor using the well-known euca2ools, an Amazon EC2 compliant REST-based client library. This module receives as input commands for a NoSQL cluster resize (in the number of running VMs). The use of euca2ools along with the creation of Amazon Machine Images (AMIs) with pre-installed versions of the supported NoSQL systems and Ganglia guarantees that Tiramola can be deployed in practically any EC2-compliant IaaS cloud.

Cluster coordinator. The orchestration of newly commissioned or freed resources from the NoSQL cluster is performed with the remote execution of shell scripts and the injection of automatically created NoSQL-specific configuration files to each VM. A high-level "start cluster", "add NoSQL node(s)" and "remove NoSQL node(s)" command is thus translated to a workflow of the aforementioned primitives. The implementation ensured that each step has succeeded before moving to the next one, using applicable time-outs. The framework has already [1] successfully incorporated three popular NoSQL systems that exhibit elastic behavior: HBase, Cassandra and Riak. The system is extensible enough to include more engines that support elastic operations by implementing the system's abstract primitives in the Cluster Coordinator module and by including the system's binaries to the existing AMI virtual machine image. The precooked virtual machine image is available for download from the project's web site. Tiramola also strives to be robust: It periodically checkpoints and can be restarted after a failure; required state is maintained through the monitoring module as well as the underlying IaaS platform

2.7.2 Tiramola's Decision Making Module

Tiramola's decision-making module is the unit that is responsible for materializing user defined policies into cluster-resizing actions. The user policies come in the form of reward functions that can evaluate the state of the cluster, and point Tiramola towards states that are in accordance to the user's needs. The state of the cluster is

acquired by Tiramola's Monitoring module, which collects a number of metrics from both the cluster and the user, and makes them available to the decision-making module. Once a resizing action has been decided, the Cloud Management module communicates with the cloud provider as well as the virtual machines in order to modify and configure the cluster into its new state.

Tiramola models the cluster as a Markov Decision Process (MDP). The states of the MDP correspond to the current size of the cluster where *k* is the number of VMs currently in the cluster and *min* and *max* are the minimum and maximum cluster sizes. The available actions of the MDP are the resizing actions and include adding or removing pre-specified numbers of VMs, or simply leaving the cluster unmodified. If a certain resizing action would exceed the minimum or maximum cluster size if executed from a certain state, then that action is made unavailable at that state (for example if the minimum cluster size is four, an action that removes two VMs would not be available at state *s*5).

In an MDP, the rewards are the feedback of the world towards the agent that informs it how good or bad the outcome of an action was. In the case of Tiramola, the result of an action is the state of the cluster after executing that action. Therefore, the reward function was calculated using the resulting state after each transition. In order to achieve a balance between giving enough resources to satisfy the user's needs, but at the same time keeping the cost of the cluster as low as possible, the reward function generally can include both positive and negative terms. For example, a reward function that aims to direct Tiramola towards performing actions that maximize the throughput and minimize the latency, while at the same time keeping the size of the cluster as low as possible, can be in the form.

Chapter 3

3. Using Tiramola

3.1 Tiramola workflow

Tiramola is responsible for the whole workflow of the experiments. When running, Tiramola is making the following steps:

a) The NoSQL-cluster has X running nodes.

b) Tiramola is a Reinforcement Learning (RL) Agent knowing its State S_1 by retrieving cluster's metrics and decides to take an action A_1 defining if the cluster will expand, contract or stay stable.

c) Due to action A_1 the cluster ends up having Y running nodes.

b) The YCSB tool is running in machines that are the VM-clients stresses the NoSQL-cluster with constant load L_1. The duration of the load involves 2 3-minute periods with an 1-minute break between them.

c) Each time the NoSQL-cluster is stressed, Tiramola is gathering metrics from 3 sources:

i) An external ganglia system that monitors the whole OpenStack installation. (***external-ganglia-metrics***)

ii) An internal ganglia system that is installed in all the NoSQL nodes. (***internal-ganglia-metrics***)

iii) Metrics reported from the YCSB running in clients. Each time YCSB-load ends, a report with metrics is generated. (***YCSB-metrics***)

d) When the whole YCSB-session ends, Tiramola is taking into consideration only the metrics gathered from the 2nd 3-minute period of YCSB-load. The 1st period considered as a warm-up.

e) Tiramola's Decision Making module defines the exact new State S_2 based on all retrieved metrics.

f) Tiramola is getting a reward R for selecting the Action A_1, based on the reward-function that is user-defined.

g) Tiramola updates the value of the State S_1 and the Action-values, known as Q-Values of the corresponding Actions based on the reward and runs the splitting algorithm, if such involved by the selected model.

The iteration ends and the system is starting again from (a), where the NoSQL-cluster now has Y running nodes, Tiramola is in State S_2 and going to decide to take Action A_2 and so on...

We will call the whole iteration a "time-step", which lasts about 10' if no extra delays happen.

3.2 Type of load

YCSB always stresses the NoSQL-cluster with constant load L_x. The load that is stated as sinusoidal in previous works that benchmark Tiramola [4 - 7] is the overall image of all different L_x loads. So, it is not a continuous sinusoidal load, but has distinct values.

In this work we are going to stress the NoSQL-cluster with the same kind of sinusoidal load when we evaluate the parameters of the MDP-DT model. Until now, all previous works [1 - 7] benchmarked Tiramola by running sinusoidal load against the NoSQL cluster. There is no prior knowledge on using Tiramola to change the size of a NoSQL-cluster that is stressed under unpredictable load, thus we are going to experiment on that case.

3.3 Modes of the Decision Making module

Tiramola is composed by 4 modules: Monitoring, Cluster Coordinator, Cloud Management and Decision Making [4]. The latter is the "brain" of Tiramola. It defines the whole State Space, the permissible Actions and the way that Tiramola evaluates each Reward and decides its next Action. Most changes and improvements on Tiramola [1 - 5] are related with this module and so did the last work [7].

In this work, we use the most recent version of Tiramola as described in [7]. In this version the Decision Making module has 4 different modes that correspond to the implementation of 4 different algorithms. The user decides which one of them to use and defines it in a properties file:

i) Q-Learning (Q)

ii) Markov Decision Process (MDP)

iii) Q-Learning with Decision Trees (Q-DT)

iv) Markov Decision Process with Decision Trees (MDP-DT)

3.4 Available metrics

As described in 3.1 (Tiramola workflow), Tiramola retrieves metrics about the NoSQL cluster during each time-step. The Monitoring module retrieves metrics from 3 sources:

i) An external ganglia system that monitors the whole OpenStack installation.(***external-ganglia-metrics***)

ii) An internal ganglia system that is installed in all the NoSQL nodes. (***internal-ganglia-metrics***)

iii) Metrics reported from the YCSB running in clients. Each time a YCSB-load ends, a report with metrics is generated. (***YCSB-metrics***)

The metrics is the most crucial part, because they define the state of the NoSQL-cluster and thus the environment of the Tiramola-agent. By retrieving these metrics the agent defines its State. The following tables contain all the 44 metrics that are available from the 3 sources along with a brief description.

IaaS metrics (external Ganglia)	
cpu	cpu of the whole user's system
number_of_threads	number of threads used by all VMs (NoSQL-cluster and clients)
read_io_reqs	read io requests of the cluster (NoSQL-cluster and clients)
write_io_reqs	write io requests of the cluster (NoSQL-cluster and clients)

Table 3.1: *IaaS metrics*

NoSQL-cluster metrics (internal Ganglia)	
bytes_in	bytes flowing into the cluster
bytes_out	bytes flowing out of the cluster
cpu_idle	percentage of cpu that is not used
cpu_nice	percentage of CPU cycles spent on nice processes
cpu_system	Percentage of CPU cycles spent in non-user mode
cpu_user	Percentage of CPU cycles spent in user mode
cpu_wio	Percentage of CPU cycles spent waiting for I/O
disk_free	The amount of the HDD that is free
load_fifteen	Reported system load, averaged over fifteen minutes
load_five	Reported system load, averaged over five minutes

load_one	Reported system load, averaged over one minute
mem_buffers	Amount of memory allocated to system buffers
mem_cached	Amount of memory allocated to cached data
mem_free	Amount of free memory
mem_shared	Amount of memory occupied by processes
mem_total	Total amount of physical memory
part_max_used	Maximum percent used for all partitions
pkts_in	Packets in per second
pkts_out	Packets out per second
proc_run	Total number of running processes
proc_total	Total number of processes

Table 3.2: *NoSQL cluster metrics*

YCSB-metrics	
%_read_load	Percentage of read load
incoming_load	Amount of the whole load that YCSB sends
read_latency	Latency of the read queries
read_throughput	Throughput of the read queries
total_throughput	Total throughput for all queries (read, update or delete)
update_latency	Latency of the update queries
update_throughput	Throughput of the update queries

Table 3.3: *YCSB client metrics*

Tiramola-made metrics	
number_of_VMs	Number of VMs of the NoSQL-cluster (with master, if NoSQL has master-slave arch)
RAM_size	Mean RAM size of all slaves of NoSQL-cluster
number_of_CPUs	Mean number of CPUs of all slaves of NoSQL-cluster
storage_capacity	Mean amount of HDD storage capacity of all slaves of NoSQL-cluster
io_reqs	= read_io_reqs + write_io_reqs
%_free_RAM	= mem_free / mem_total
%_cached_RAM	= mem_cached / mem_total
%_CPU_usage	= 100 - cpu_idle
%_read_throughput	= read_throughput / total_throughput
total_latency	= (READ_LATENCY * READ_THROUGHPUT + UPDATE_LATENCY * UPDATE_THROUGHPUT) / (READ_THROUGHPUT + UPDATE_THROUGHPUT)
next_load	= 2 * current_load - last_load. It is a simple linear forecasting of the next load.
network_usage	= bytes_in + bytes_out

Table 3.4: *Combined metrics*

3.5 State Spaces

3.5.1 State Spaces for beginners in Reinforcement Learning

One of the most well-known examples for beginners in Reinforcement Learning (RL) involves an agent that moves around in a grid, which has 12 squares defining 12 different states for the RL-agent. One of the states is usually the goal and gives the maximum Reward and another one is something like trap giving minimum or negative

Reward. The available Actions of the agent and all the other possible Rewards are defined and the student has all the data for solving the first RL-problem.

Figure 3.1: *State-space of RL-exercise for beginners*

During the next lessons an RL-student will deal with more RL-problems having similar State Spaces defined as 4X4, or 4X5 etc. grids. Each State Space is defined in Cartesian-like names, chess-like or each State may have a serial number.

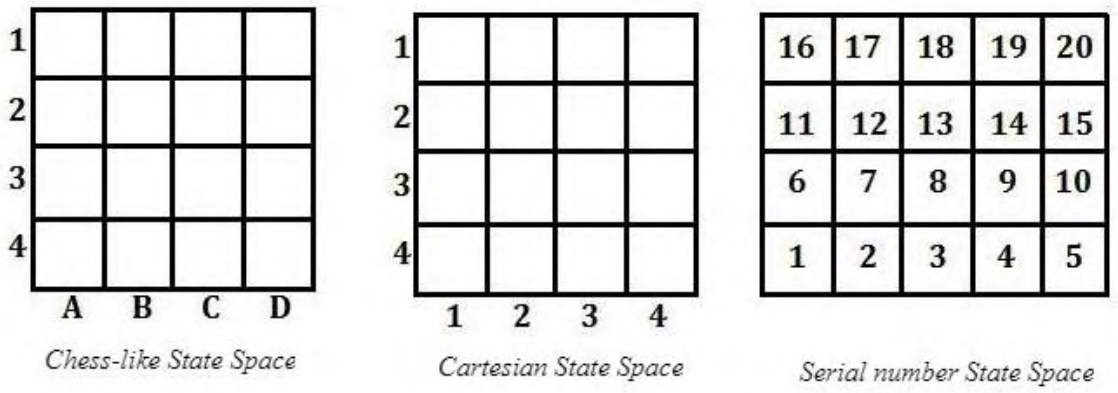

Figure 3.2: *Simple State Spaces in Reinforcement Learning*

These all are simple cases for exercises, where the Agent is usually allowed to decide between the Actions: Up, Down, Left and Right. The grid is like a map and the RL-agent is just a walker in the map, having a certain goal in each problem. It is also clear that the State Space in each of the above has 2 Dimensions and each Dimension has units specified by each square.

3.5.2 State Spaces in Tiramola

During every time-step, the NoSQL-cluster is stressed with load. When the stressing is over, Tiramola receives the 44 metrics which describe NoSQL-cluster's reaction. As stated earlier, the Tiramola-agent can use them to define its State, but first we should decide which one of them the agent will use. For instance, if we let the agent

use them all, then a State Space of 44 Dimensions will be available, which is surely a too complicated State Space.

We want the Tiramola-agent to be able to add or remove VMs, so the available Actions are “Add X Vms”, “Remove X Vms”, “No Action”, with X being 1, 2, or whatever the user decides. Thus, it is obvious that one of the State Space’s Dimensions will be the number_of_VMs, but one Dimension cannot be enough. We should define at least one more.

Part of this work will be to define which of the rest 43 metrics are the most appropriate to be used as Dimensions of the State Space. Seeing this problem from a different point of view, we are going to evaluate which if the 43 metrics are the most suitable to be used from the MDP-DT algorithm as splitting parameters. The reason for this equivalence of these 3 terms is that for the Tiramola’s Decision Making Module the metrics of the cluster, the Dimensions of the State Space and the splitting parameters are all the same thing. Tiramola gets the cluster metrics, uses some of them as splitting parameters to create a State Space and from the State Space point of view these metrics / splitting parameters are its Dimensions.

3.5.2.1 State Space in Q and MDP modes

The Q and MDP algorithms create a State Space that is more affected by the user than the Q-DT and MDP-DT ones. In both Q and MDP algorithms, we modify a .json file where we define which of the metrics will be considered as parameters of the State Space. To make it clear, a metric and a parameter is the same, but a metric gets its value from the cluster’s behavior, while a parameter has its values defined by the user. Taking into account the previous descriptions of the State Space, a parameter is a Dimension of the State Space and its user-defined values are the units of that Dimension. The user selects number_of_VMs as a parameter and most of the times next_load is selected as the second parameter.

Having these 2 parameters, the Tiramola-agent can now define the State Space. We can think of it like a grid we often see in Reinforcement Learning problems. Tiramola’s Dimensions of the State Space are defined by number_of_VMs values and by next_load value-range. Both of these parameters and their values are defined by the user. For instance, if we have a NoSQL-cluster that can contract or expand from 4 VMs to 10 VMs, the obvious choice for the values of number_of_VMs is 4, 5, 6, 7, 8, 9 and 10. Also, we will define values for the next_load that will be converted to value-ranges. For instance if we select [1000, 5000, 10000, 22000, 35000], the Tiramola-agent will convert it to 4 value-ranges: [1000, 5000], [5000, 10000], [10000, 22000], and [22000, 35000]. Eventually, the State Space will look like the grids of the exercises for RL-beginners and will be composed from a number of states equal to the product of number_of_VMs

values by next_load value-ranges, meaning 7 * 4 = 28 States. The States will be created based on these two parameters as follows:
S0: [4, [1000, 5000]], S1: [5, [1000, 5000]] ...etc... S6: [10, [1000, 5000]], S7: [4, [1000, 5000]], S8: [5, [5000, 10000]] ...etc… S13: [10, [5000, 10000]]...etc, until S27.

Now, as described in steps (b) and (e) in 3.1 (Tiramola workflow), the Tiramola-agent by retrieving the metrics can define its State. For instance, if during the load-time the NoSQL-cluster has 5 VMs (number_of_VMs = 5) and the next_load metric has value of 3,000, then Tiramola-agent knows that it is in S_1 State.

Leaving aside that Q and MDP have a huge difference in their updating algorithm and are considered as model-free and model-based RL approaches respectively, seeing these algorithms from the Tiramola-agent point of view, they have a great similarity. Each of them has its State Space fully defined by the user.

3.5.2.2 State Space in Q-DT and MDP-DT modes

In Q-DT and MDP-DT algorithms, the user defines the metrics that will be used as parameters for the State Space, but does not define their values or value-ranges. In fact the user may not even know what the values of each parameter are at all. After each time-step, Q-DT or MDP-DT are updating the values of the current State and the Action-values (Q-Values) of every Action based on their updating algorithm and the Reward. Then, they evaluate whether they can split the current State or not, based on the accumulated experience. When a splitting algorithm runs, each of the (user-defined) parameters are being checked if they are suitable to split the current State into two new States. Each of the parameters is checked separately and the user-selected statistical test returns a value. By using that value, the splitting algorithm calculates if a parameter is suitable for splitting the current State. The lower the value, the bigger the probability of doing the split. If more than 1 parameters are suitable for a split, the algorithm selects the most probable one.

The original algorithm of the Decision Trees [18] defines that the Decision Tree starts with only one node, the root. While the experiment is running and especially during the training period, the splits happen and the Decision Tree grows. Each Decision Node is defined by a parameter and a specific value and points to 2 Leafs, either a State or a Decision Node, which are the Children. Only the last level of the Decision Tree is composed exclusively by States.

Starting the Decision Tree / State Space with only one Leaf / State, means we have no clue about the Tiramola-environment. Considering that such a case does not exist, we can boost the Decision Tree with some initial Leafs. As in Q and MDP algorithms, the most suitable metric to be a parameter for the initiation of the Decision Tree is the number_of_VMs, complying with the fact that we surely know the possible

number of VMs the NoSQL-cluster has. As the preliminary experiments showed, even if we start a session with only one starting-State, Q-DT and MDP-DT algorithms will surely find the exact possible number of VMs. All the splits based on all different number_of_VMs values will eventually happen and Tiramola will have perfect knowledge of its environment regarding the number of VMs. Considering that fact, there is no need in consuming time-steps for this to happen when we can give that information to the algorithm from the beginning.

3.6 Using the last version of Tiramola

In the work that implemented the last version of Tiramola [7] we can clearly distinguish that the full-model with Decision Trees, the MDP-DT performs better than the other models. Also, a lot of effort was given in experimenting with several parameters either of the MDP algorithm or of the Decision Trees and it was made clear which values are preferred for each one of the model-related parameters.

In this work, the primary target is to run Tiramola and study its behavior while the NoSQL-cluster is stressed by unpredictable load. In all previous works [1 - 7] Tiramola was always stressed by periodical load. For doing this challenging task, Tiramola should work at its best and even though the last work [7] defined the best values for the most model-related parameters (epsilon, initial_qvalues, discount, min_measurements, splitting criterion, statistical test, model, update algorithm), there are still some more to clarify before using it.

In this work we will define which metrics are better to use as parameters for the Decision Trees and we will try to explain why there are better or worse parameters. To do this, we will study the metrics of an HBase-cluster in order to describe its behavior when it is stressed. For doing the experiments more effectively, we will separate Tiramola's workflow. At first we will get the metrics stressing the NoSQL-cluster by all possible loads and for all possible number_of_VMs. Then we will study these metrics to abstractly define the behavior of the NoSQL cluster. In the end, we can run the Virtual Tiramola where the Decision Making procedure will retrieve in each time-step the previously retrieved metrics. This will give us the freedom to run many more experiments/sessions in less time, because the time-step will last some milliseconds, instead of 10 minutes.

Chapter 4

ANALYZING HBASE CLUSTER METRICS

4.1 Objective

In order to decide the proper size of a NoSQL cluster, the metrics we retrieve while it is up and running and stressed under any kind of load are the raw material that is used by any system. So does Tiramola in order to automate elasticity of NoSQL clusters. The great difference is the way each system exploits these metrics.

In this chapter, we are going to study the behaviour of these metrics. Later, we will combine this kind of knowledge with the way Tiramola is exploiting them, in order to come up with a conclusion. At the first part of experiments, we study the behavior of 10 different HBase metrics while linear increasing load runs against the HBase cluster. In the second part, we study the metrics' behavior, while the cluster is under constant load.

The total number of metrics is 44, but we can divide them to the direct metrics and indirect ones. By studying the latter, we practically studying the direct ones too. By looking at the tables 3.1, 3.2, 3.3 and 3.4, we can see that studying the metrics: %_CPU_usage, total_latency, network_usage, load_one, total_throughput, io_reqs, cpu_wio, %_free_RAM, disk_free and %_cached_RAM, we also manage to study another 11 metrics: read_io_reqs, write_io_reqs, bytes_in, bytes_out, cpu_idle, mem_free, mem_total, read_latency, read_throughput, update_latency, update_throughput. In addition to that, number_of_vms and next_load do not require further study, and load_fifteen and load_five are useless, because each time the clients stress the HBase cluster they do it for less than 5 minutes. Consequently, we manage to study the behavior of 21 out of 40 metrics that need to and can be studied, and more importantly, we cover all areas of metrics that define a running cluster of machines: CPU usage, RAM usage, disk usage and network usage, thus having an adequate view.

4.2 Experiments pt. 1: Linear increasing load

We have 13 VMs available for experiments, so we will use 1 + 8 VMs for the HBase-cluster and 4 client-VMs running the YCSB tool. Selecting for the replication factor of HDFS to be 2, we set the minimum size of the HBase-cluster at 2 VMs-slaves and maximum at 8 VM-slaves and do the benchmarking against 3, 4, 5, 6, 7, 8 and 9 VMs (master included). The load we will run has range 1,000 to 25,000 reqs/sec in steps of 100 reqs/sec. While the load increases the size of the Hbase cluster reamains the same.

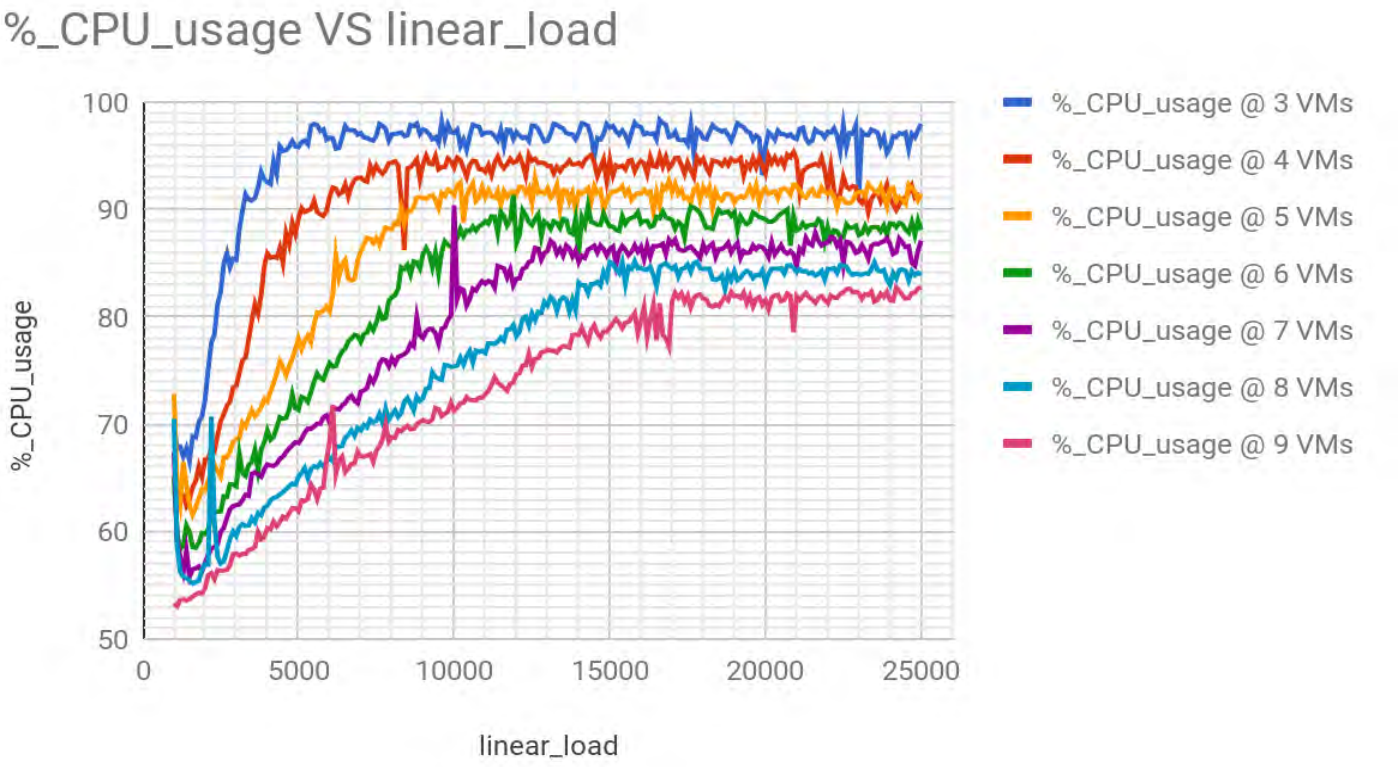

Figure 4.1: *%_CPU_usage behavior against linear increasing load*

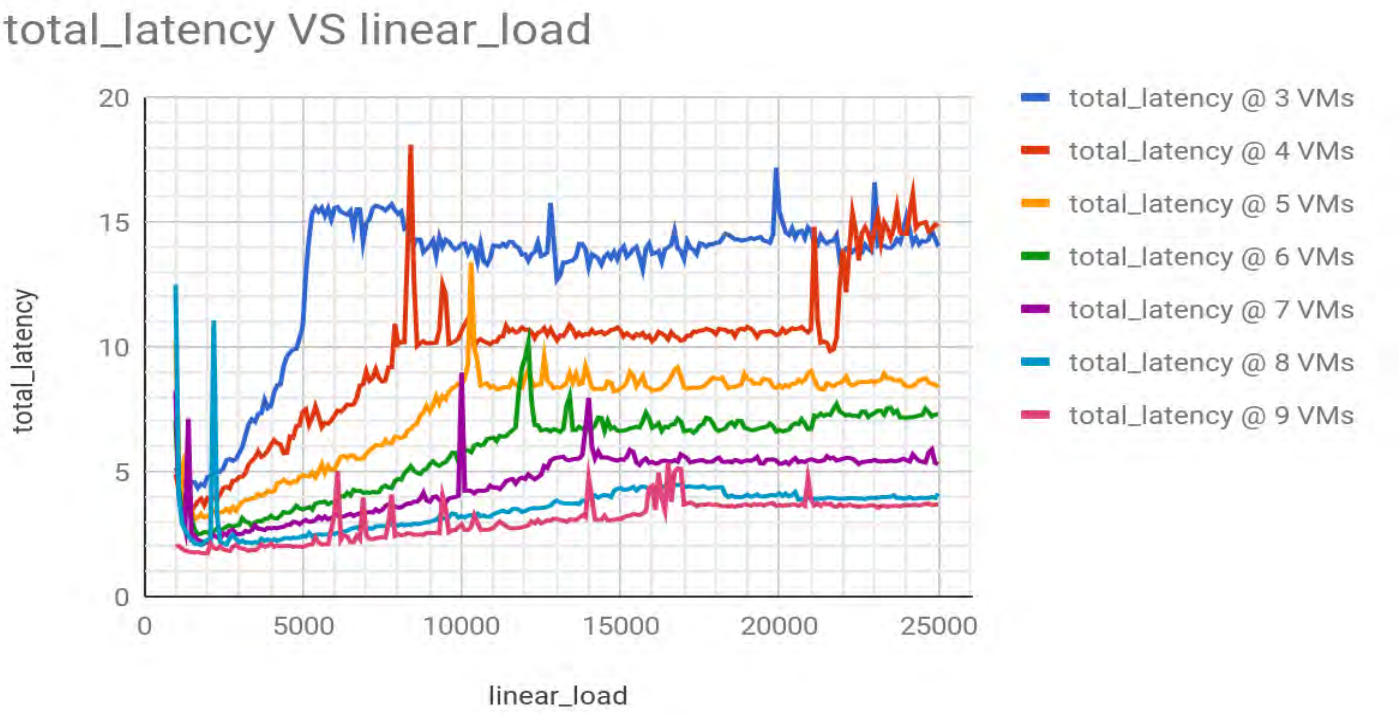

Figure 4.2: *total_latency behavior against linear increasing load*

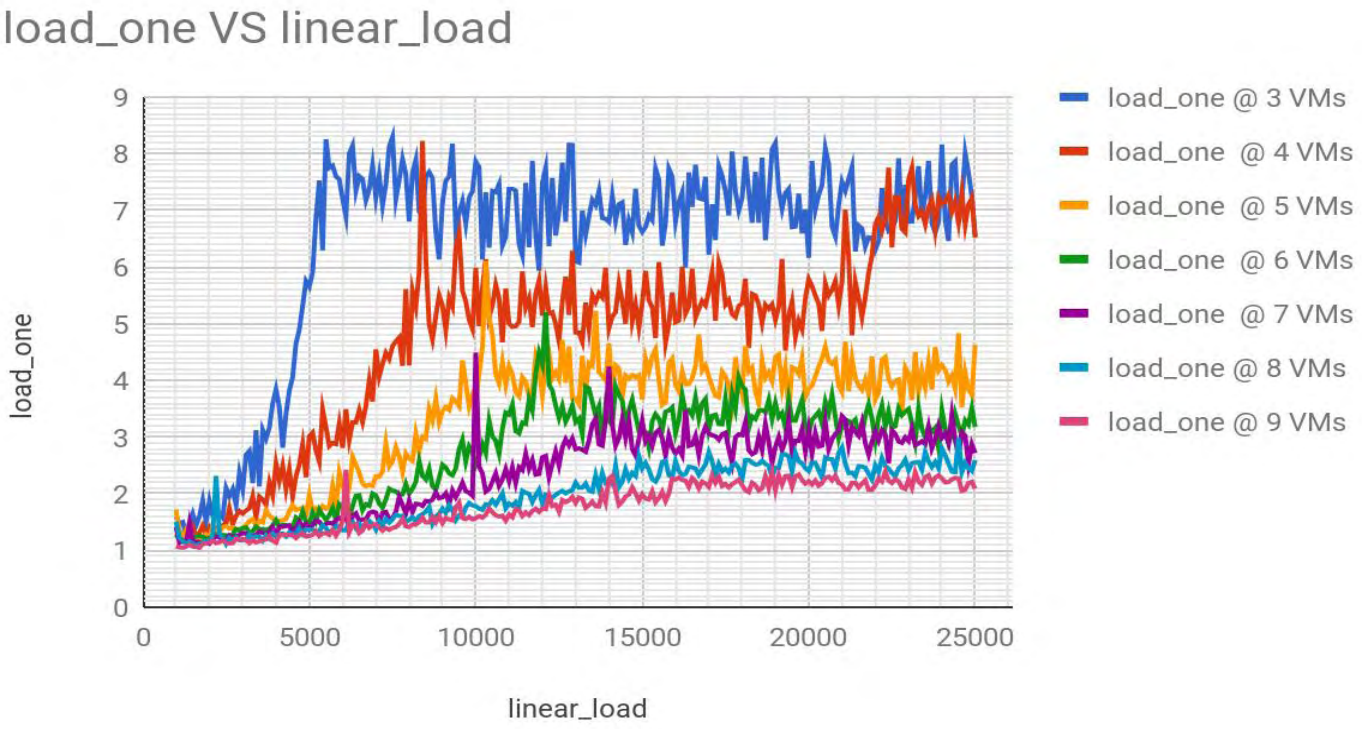

Figure 4.3: *load_one behavior against linear increasing load*

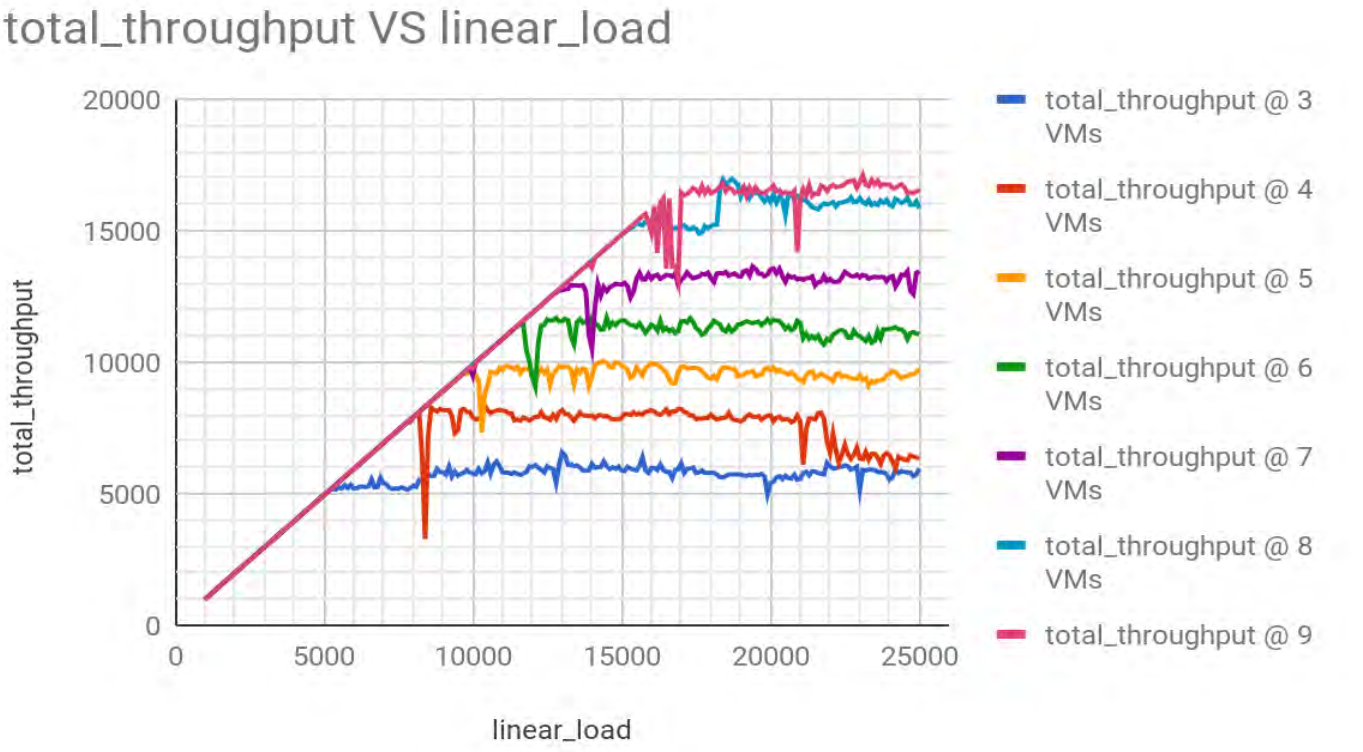

Figure 4.4: *total_throughput behavior against linear increasing load*

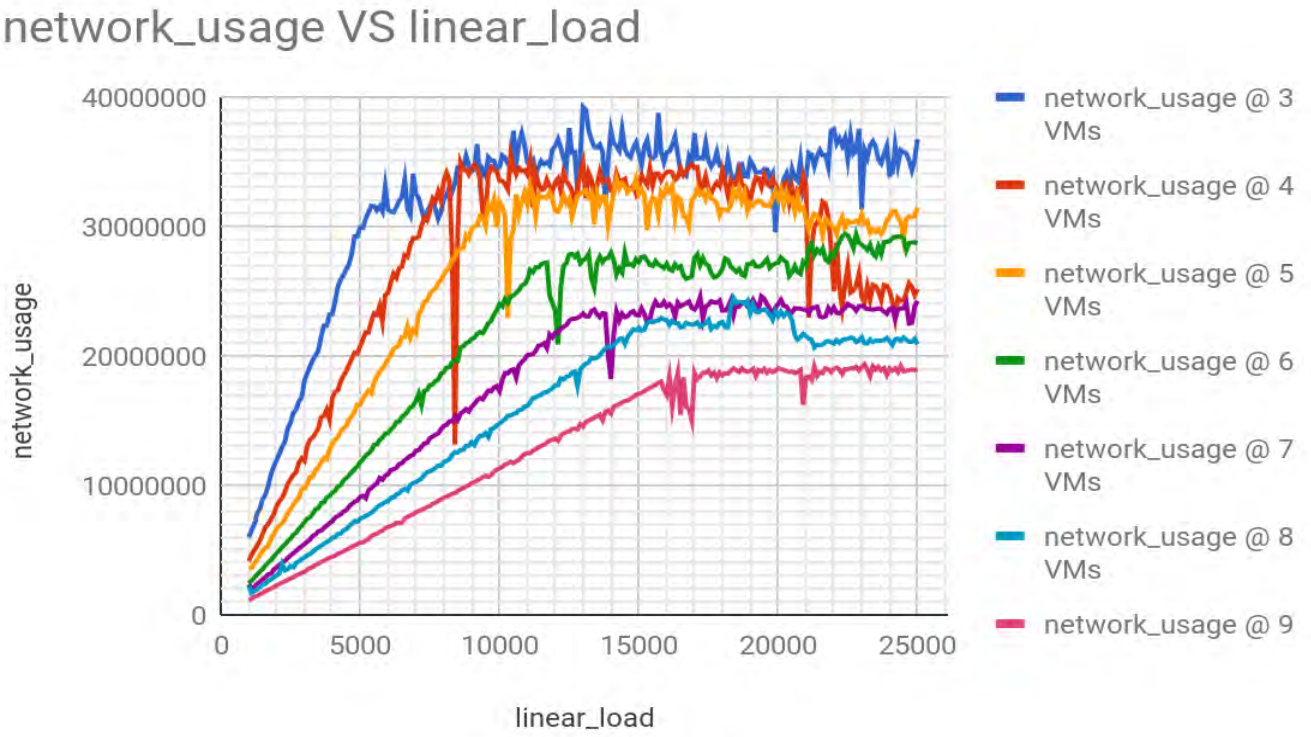

Figure 4.5: *network_usage behavior against linear increasing load*

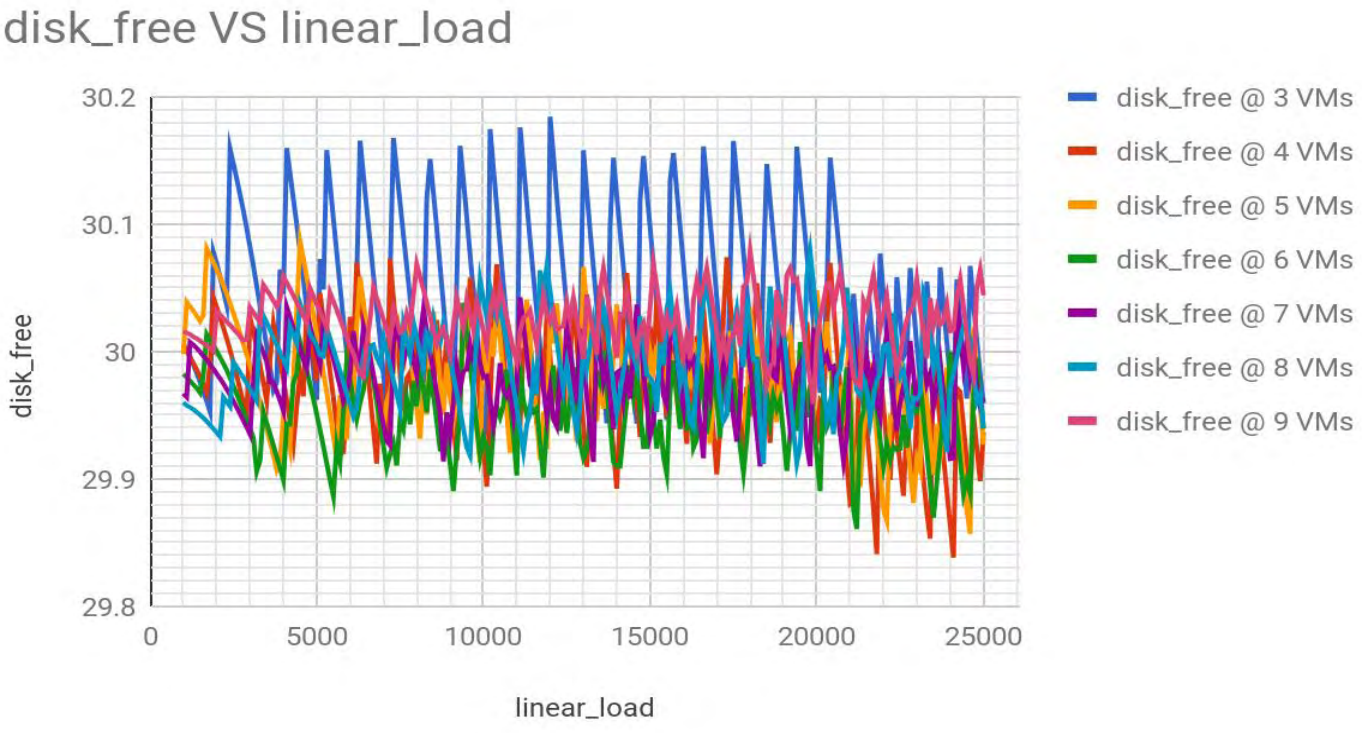

Figure 4.6: *disk_free behavior against linear increasing load*

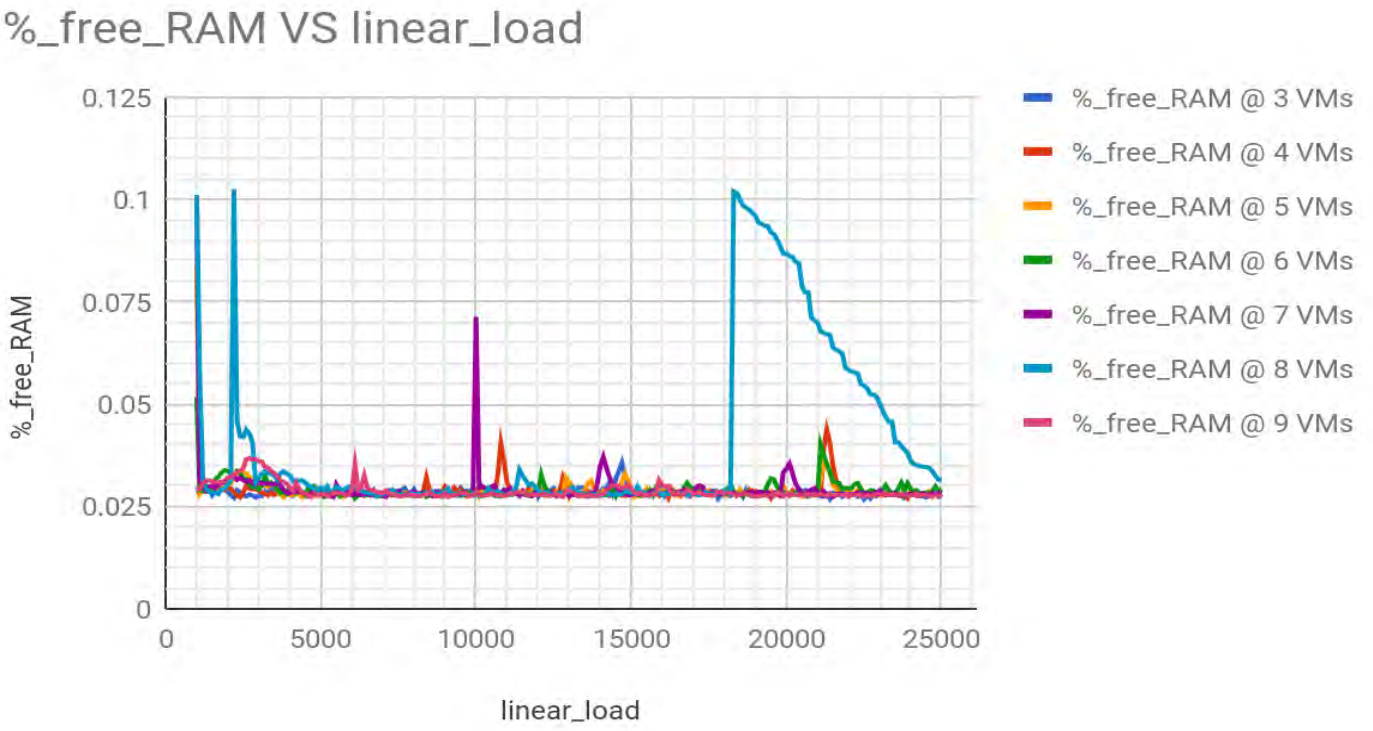

Figure 4.7: *%free_RAM behavior against linear increasing load*

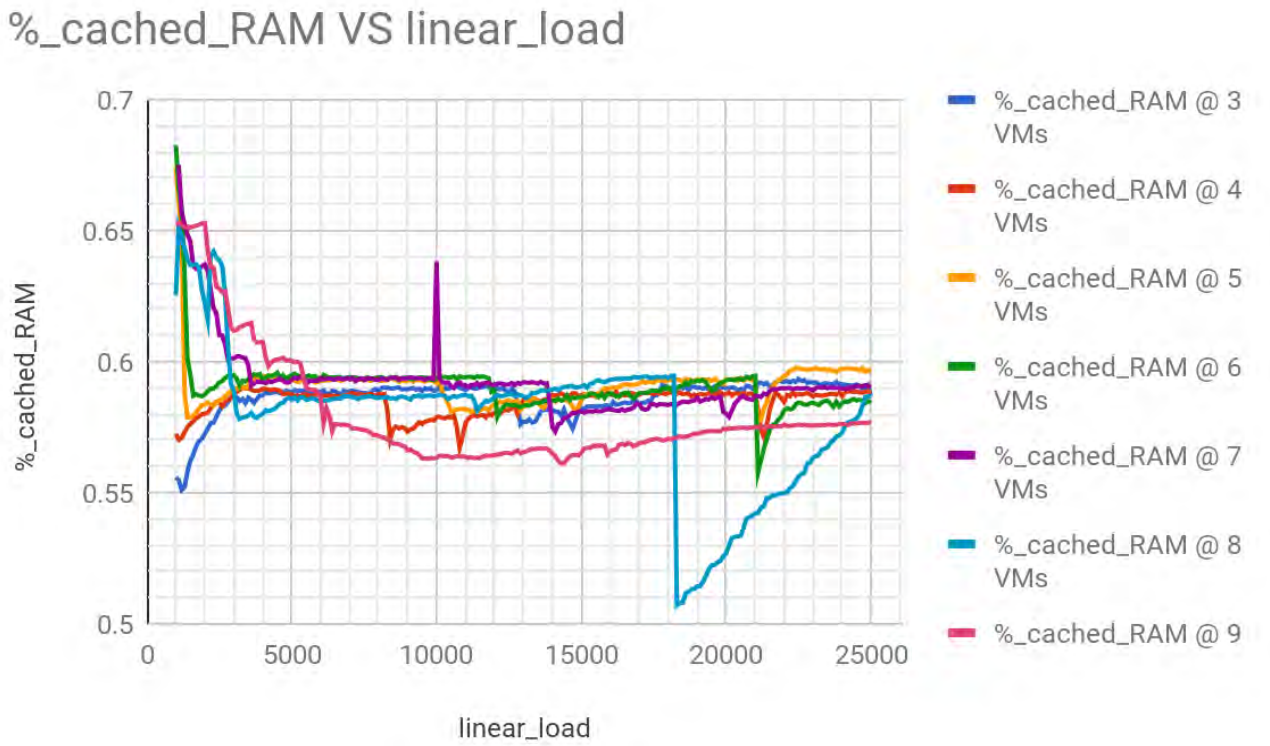

Figure 4.8: *%_cached_RAM behavior against linear increasing load*

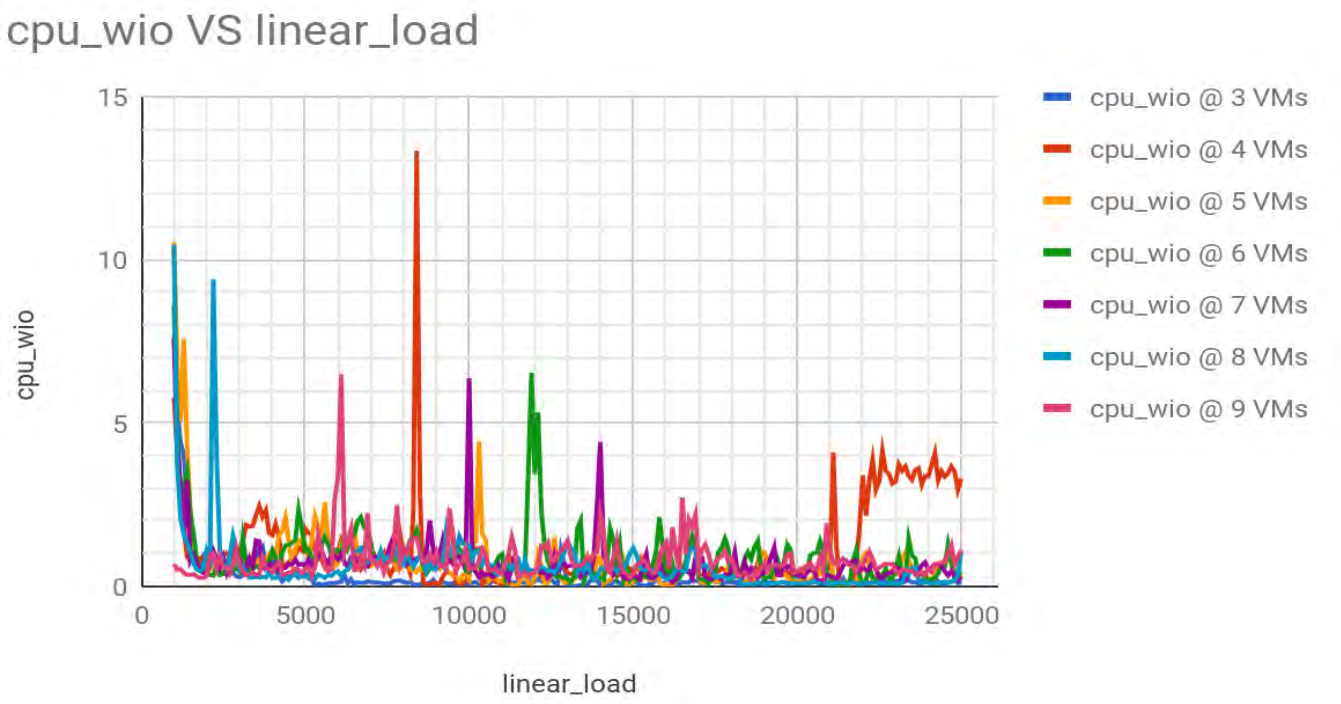

Figure 4.9: *cpu_wio behavior against linear increasing load*

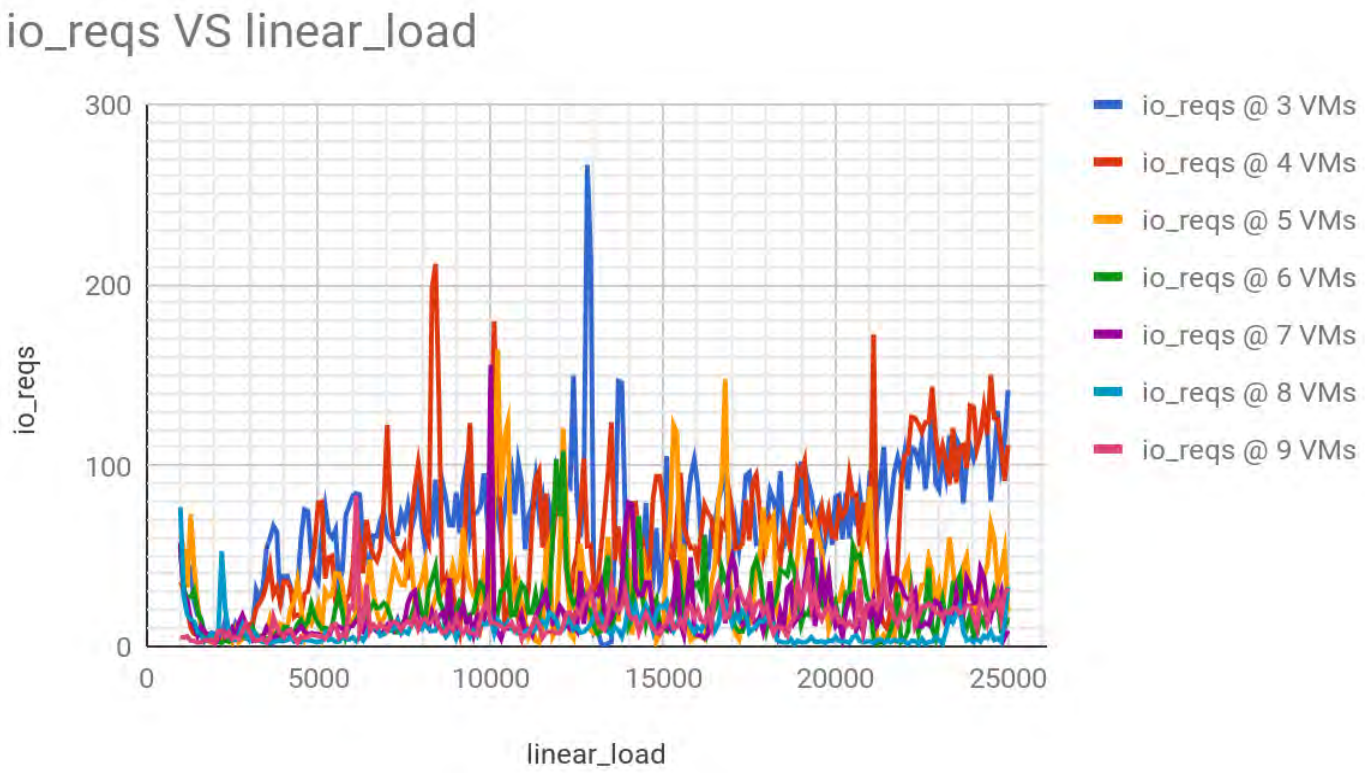

Figure 4.10: *io_reqs behavior against linear increasing load*

4.2.1 Metrics' behavior under linear load

Based on their behavior against linear increasing load, we can divide the metrics into two groups.

Group 1	Group 2
%_CPU_usage	io_reqs
total_latency	cpu_wio
network_usage	%_free_RAM
load_one	disk_free
total_throughput	%_cached_RAM

Table 4.1: *Metrics according to behavior under linear increasing load*

In the first group the metrics are increasing alongside the load, until a specific point. After that point, despite that the load continues to increase, each of the metrics seems to be almost stable. This shows that the cluster reaches the maximum of its performance and can't go higher. We can call this load, the "critical load". The critical load is different for different size of the cluster, but the same for each metric.

In the next table we show the critical load for different sizes of the cluster based on the total_throughput.

Cluster Size	Critical Load
3	5300
4	8200
5	9700
6	11600
7	12900
8	15300
9	17000

Table 4.2: *Critical load for each cluster size*

4.2.2 Conclusions about metrics behavior for linear increasing load

As it is expected, when the cluster has more nodes it also has a higher performance maximum. While the load increases the values of the metrics of the 1st group also increases. On the other hand, we cannot distinguish any pattern at all for the metrics of the second group.

4.3 Experiments pt. 2: Constant load

We will dive more into each metric's behavior by running the same load 10 times against all available HBase-cluster sizes. For each cluster size we divide the range of loads from 1000 reqs/sec until the critical (different for every cluster size) in several steps. We will show the behavior of 2 metrics, one of the 1st group, (**%_CPU_usage)** and one of the 2nd, (**io_reqs)** in graphs in order to have a view of how the metrics of each group behave under specific loads that are lower than the critical load for 3 different sizes of the HBase-cluster. For loads equal or higher than the critical, all metrics reach a global maximum, or minimum and we get no useful information by viewing such graphs.

We stressed the HBase with loads from 1000 reqs/sec until the critical one (different for every cluster size). Given that we did it for all the different available sizes of the cluster: 3, 4, 5, 6, 7, 8 and 9 VMs it means that in total there are 70 graphs. Based on the data gathered by all 70 graphs, we choose to present the 3 most representative for each respective group of metrics. For clarity purposes, we select to present the results only for 3, 6 and 9 VMs for only 2 metrics (%_CPU_usage and io_reqs), one from each group.

Figure 4.11: *%_CPU_usage @ 3VMs against constant loads: 2000, 3600 and 5000*

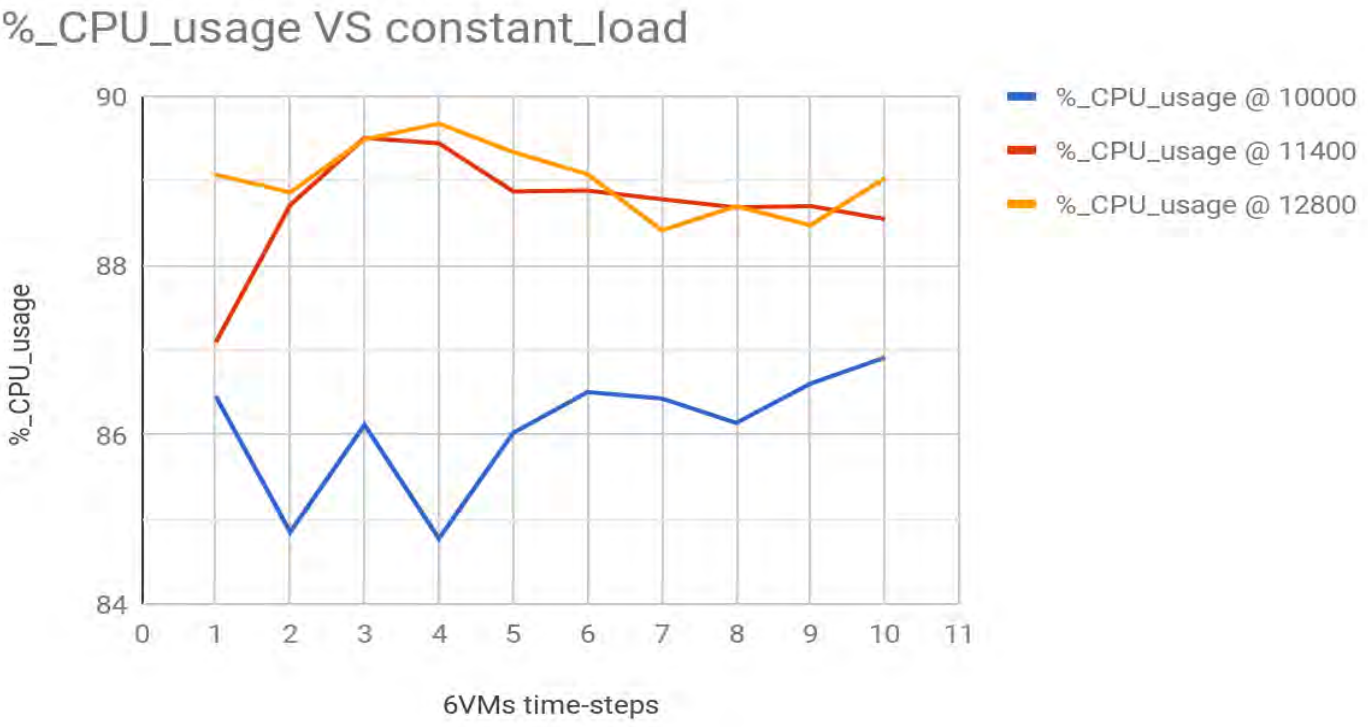

Figure 4.12: *%_CPU_usage @ 6VMs against constant loads: 10000, 11400 and 12800*

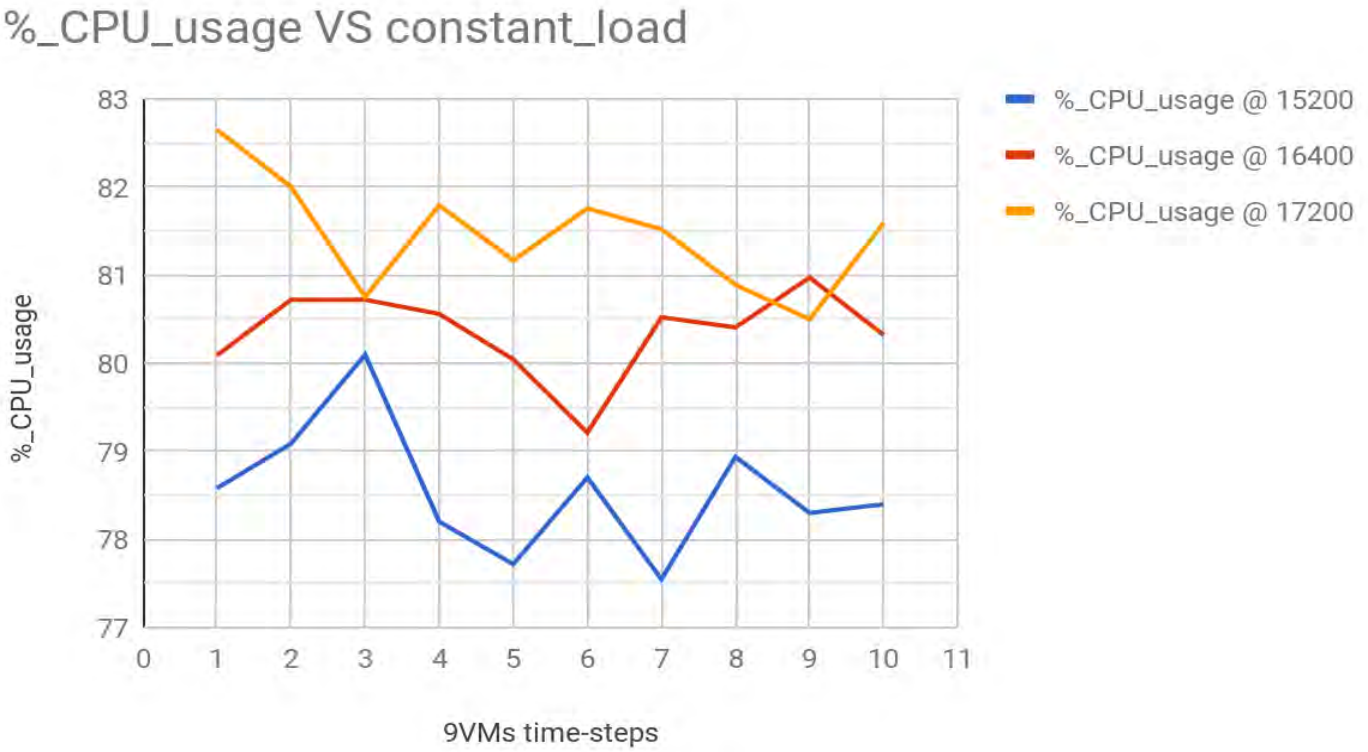

Figure 4.13: *%_CPU_usage @ 6VMs against constant loads: 15200, 16400 and 17200*

Figure 4.14: io_reqs *@ 3VMs against constant loads: 2000, 3600 and 5000*

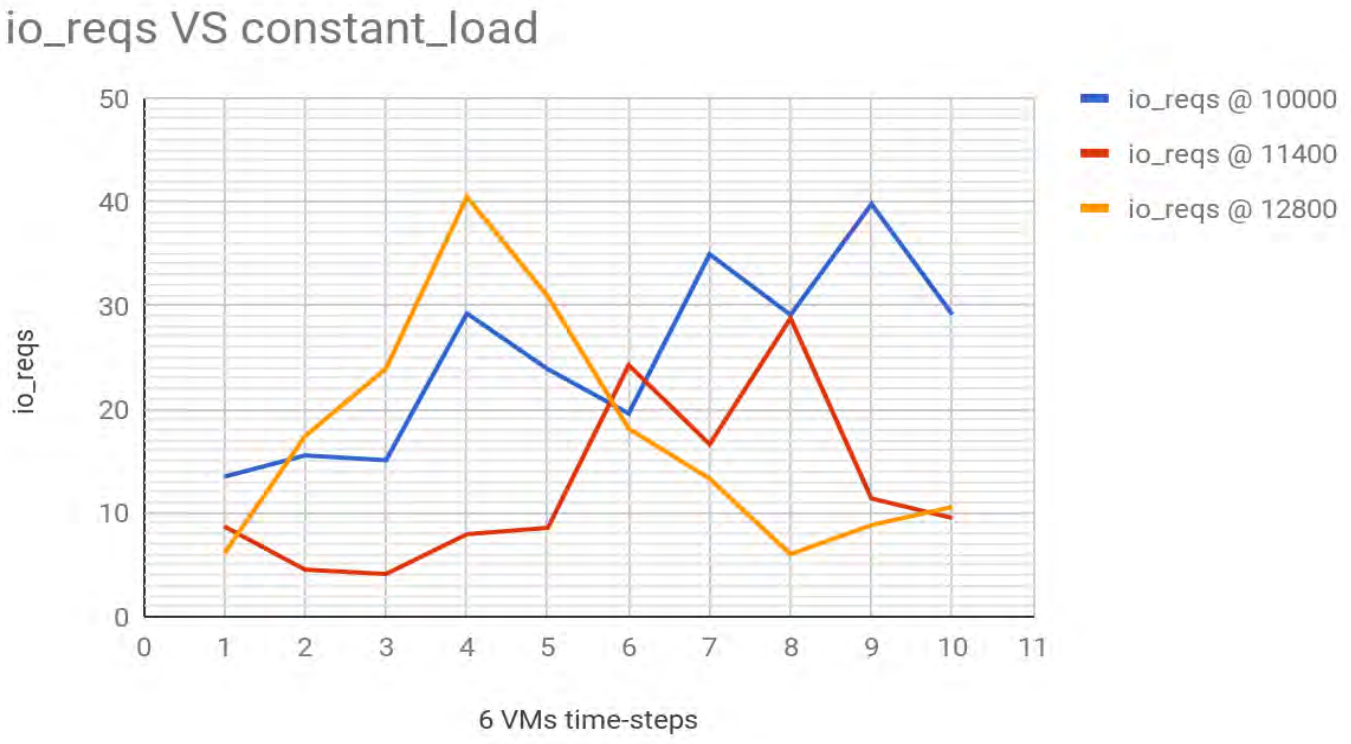

Figure 4.15: io_reqs *@ 6VMs against constant loads: 10000, 11400 and 12800*

Figure 4.16: io_reqs *@ 6VMs against constant loads: 15200, 16400 and 17200*

4.3.1 Metrics' behavior under constant load

The graphs are showing that the metrics of the 1st group, like %_CPU_usage are more stable against the same load. Also, for different loads that are lower than the critical one, they differentiate enough.

On the other hand graphs of the metrics of the 2nd group, like io_reqs are less stable against the same load. Also, for different loads that are lower than the critical one, they do not differentiate.

We already had a clue of such behavior when studying the graphs from the 1st phase of experiments (4.2). By running the same load many times and watching each metric alone, we can be more certain of how stable a metric is when the HBase cluster is stressed against the same load. Furthermore, we can compare the values of one metric when the same HBase cluster size is stressed against different loads.

In 4.3.2 we can see the average and the coefficient of variance for each metric for all cluster sizes and each metric's behavior will be even clearer.

4.3.2 Data analysis of metrics. Cluster is stressed by constant load close to critical

Load @ 3 VMs	2,000 reqs/sec		3,600 reqs/sec		5,000 reqs/sec	
	average	cf_var	average	cf_var	average	cf_var
%_CPU_usage	77.72381	0.01593	92.172	0.004844	95.85738	0.00493
load_one	1.8075	0.08843	2.978738	0.154038	4.40281	0.08295
total_throughput	1980	0.00025	3564	0.00094	4948	0.00045
total_latency	5.452527	0.02654	7.568624	0.023333	9.696665	0.01817
network_usage	1131310	0.00607	20401833	0.007231	2985658	0.02769
%_cached_RAM	0.590990	0.00201	0.588234	0.000777	0.590121	0.00047
%_free_RAM	0.028435	0.01099	0.028745	0.016131	0.02785	0.01011
cpu_wio	4.074286	0.16767	2.051905	0.184826	0.436667	0.15534
disk_free	29.94177	0.00106	30.00871	0.002268	30.00585	0.00142
io_reqs	68.62142	0.20337	143.3857	0.09652	92.28095	0.13455

Table 4.3: *avg and cf_var of metrics @ 3 VMs*

Load @ 4 VMs	4800 reqs/sec		6000 reqs/sec		7200 reqs/sec	
	average	cf_var	average	cf_var	average	cf_var
%_CPU_usage	86.70111	0.008913	90.306	0.003178	93.02809	0.00803
load_one	2.412111	0.087961	3.001302	0.070168	4.468651	0.16461
total_throughput	4750	0.000596	5936	0.000697	7061	0.01753
total_latency	6.074851	0.014771	7.055016	0.03071	9.660051	0.17736
network_usage	1861754	0.010418	2341309	0.010243	2779146	0.03125
%_cached_RAM	0.588928	0.000817	0.588384	0.000824	0.576348	0.00369
%_free_RAM	0.028313	0.012940	0.028355	0.012172	0.028955	0.03457
cpu_wio	0.895238	0.298173	0.624444	0.434906	0.592857	1.21474
disk_free	29.99079	0.000794	29.97785	0.001515	29.98571	0.00219
io_reqs	33.40317	0.358443	41.8937	0.472	78.44285	1.25439

Table 4.4: *avg and cf_var of metrics @ 4 VMs*

Load @ 5 VMs	7,200 reqs/sec		8,600 reqs/sec		10,000 reqs/sec	
	average	**cf_var**	**average**	**cf_var**	**average**	**cf_var**
%_CPU_usage	87.182619	0.008003	90.51059	0.008356	91.64190	0.003329
load_one	2.503595	0.06692	3.330357	0.05715	4.24244	0.066959
total_throughput	7127	0.000309	8510	0.000636	9421	0.012224
total_latency	6.325773	0.020495	8.023636	0.028752	8.604582	0.018364
network_usage	22718157	0.016769	2690874	0.005056	3029105	0.015186
%_cached_RAM	0.583332	0.001211	0.586646	0.002060	0.597155	0.000303
%_free_RAM	0.028362	0.014757	0.028554	0.007577	0.027915	0.007157
cpu_wio	0.457857	0.614129	0.414286	0.572392	0.553929	0.477591
disk_free	29.971383	0.001554	29.98094	0.001463	29.93228	0.000889
io_reqs	19.715476	0.573954	33.18928	0.467146	37.65476	0.36236

Table 4.5: *avg and cf_var of metrics @ 5 VMs*

Load @ 6 VMs	10,000 reqs/sec		11,400 reqs/sec		12,800 reqs/sec	
	average	**cf_var**	**average**	**cf_var**	**average**	**cf_var**
%_CPU_usage	86.081048	0.007915	88.724381	0.007022	89.015524	0.004424
load_one	2.564143	0.060328	3.186514	0.062241	3.328762	0.077475
total_throughput	9886	0.002148	11275	0.001143	11612	0.014052
total_latency	5.985481	0.022798	6.208749	0.021719	6.687669	0.021069
network_usage	25644203	0.00551	24697072	0.020199	25474216	0.014908
%_cached_RAM	0.592324	0.000702	0.563658	0.011544	0.543696	0.006677
%_free_RAM	0.028158	0.010977	0.035834	0.097463	0.028886	0.016598
cpu_wio	0.868476	0.335025	0.232571	0.624887	0.342667	0.749432
disk_free	29.931784	0.00079	30.011533	0.00112	30.011155	0.00125
io_reqs	24.981905	0.339644	12.446667	0.629897	17.566667	0.610754

Table 4.6: *avg and cf_var of metrics @ 6 VMs*

Load @ 7 VMs	12,800 reqs/sec		14,000 reqs/sec		15,000 reqs/sec	
	average	**cf_var**	**average**	**cf_var**	**average**	**cf_var**
%_CPU_usage	85.088651	0.006111	86.370397	0.006523	86.582778	0.004482
load_one	2.569897	0.078018	2.779032	0.057983	2.786476	0.064373
total_throughput	12666	0.000437	13480	0.012062	13219	0.008765
total_latency	4.829598	0.018814	5.425167	0.019126	5.573686	0.016596
network_usage	23284062	0.006517	24568631	0.019852	25994527	0.011533
%_cached_RAM	0.586181	0.000365	0.587606	0.000925	0.593965	0.000456
%_free_RAM	0.028107	0.007338	0.028469	0.011896	0.02794	0.010601
cpu_wio	0.359841	0.237559	0.535079	0.701306	0.544841	0.574781
disk_free	29.965803	0.000528	29.964808	0.000392	29.970753	0.00106
io_reqs	11.361111	0.44737	18.285714	0.972801	22.200794	0.455183

Table 4.7: *avg and cf_var of metrics @ 7 VMs*

Load @ 8 VMs	14,000 reqs/sec		15,200 reqs/sec		16,400 reqs/sec	
	average	**cf_var**	**average**	**cf_var**	**average**	**cf_var**
%_CPU_usage	82.641769	0.006251	84.26966	0.006423	83.852245	0.005462
load_one	2.217776	0.059248	2.399449	0.048643	2.344	0.052588
total_throughput	13857	0.000677	14909	0.005279	14992	0.006868
total_latency	4.274929	0.027742	4.625379	0.014695	4.611362	0.004256
network_usage	23772741	0.003649	25512127	0.007383	25633135	0.009593
%_cached_RAM	0.584463	0.001431	0.586576	0.000611	0.587318	0.000617
%_free_RAM	0.028802	0.010101	0.028724	0.013494	0.029026	0.013826
cpu_wio	0.451293	0.251733	0.495714	0.720546	0.283946	0.197408
disk_free	29.985271	0.000377	29.978847	0.000817	29.979841	0.001001
io_reqs	15.055102	0.321264	13.705442	0.488938	11.678912	0.391404

Table 4.8: *avg and cf_var of metrics @ 8 VMs*

Load @ 9 VMs	15,200 reqs/sec		16,400 reqs/sec		17,200 reqs/sec	
	average	**cf_var**	**average**	**cf_var**	**average**	**cf_var**
%_CPU_usage	78.55779	0.008779	80.356607	0.005831	81.462381	0.007533
load_one	1.966202	0.059277	2.07722	0.036486	2.126738	0.027519
total_throughput	15048	0.000984	16232	0.000617	16946	0.003409
total_latency	3.112685	0.036	3.301108	0.021883	3.483197	0.020624
network_usage	17434238	0.011549	18857000	0.003708	19638832	0.007305
%_cached_RAM	0.50802	0.021791	0.547197	0.017802	0.573302	0.007521
%_free_RAM	0.106464	0.104371	0.066956	0.145066	0.04171	0.081629
cpu_wio	0.436726	1.232836	0.321369	0.562276	0.603929	0.673864
disk_free	30.01915	0.001249	30.007959	0.001275	30.010566	0.00142
io_reqs	4.396429	0.966439	6.407143	0.720265	12.529762	0.443305

Table 4.9: *avg and cf_var of metrics @ 9 VMs*

4.3.3 Conclusions about metrics behavior under constant load

The experiments with constant load lead us to two conclusions. The first is about the variation of each metric and the second is how we can distinguish the different environment-status of the HBase-cluster.

cpu_wio and io_reqs have higher variation than all the others. We run the same load 10 times in a row against the same size HBase-cluster, but cpu_wio's and io_reqs' values are quite different each time, resulting in a high value in the coefficient of variation (cf_var). cf_var shows the level of variation, is a measure of relative variability and free of measurement units, so we can use it as a comparator among all metrics.

%_cached_RAM, %_free_RAM and disk_free have low variation, but their values do not differ when the load or the HBase-cluster size change. If we take only them into account we get the impression that there is no difference in HBase-cluster's behavior whether there are changes in the load or the size of the HBase-cluster. Such a behavior is somewhat strange and not helpful as a Dimension of the State Space.

On the other hand, all the metrics of the 1st group have a low coefficient of variation when the HBase-cluster has the same size and is stressed under the same load. When the size of the cluster or the load change, the values of the metrics change. When the load is above the critical load each of these metrics have almost the same value, showing that the cluster is performing at its maximum.

Chapter 5

EXPERIMENTAL RESULTS

5.1 Objective

In this round of experiments we use the last version of Tiramola and focus on the MDP-DT algorithm. This algorithm is already evaluated as the optimal among MDP, Q and Q-DT in previous work. Also, the optimal values for its parameters, have already been defined [6] (5.2.3).

In chapter 4 we studied the metrics of the HBase cluster and led into conclusions about their behavior and assumptions about their role as Dimensions of a State Space (4.3.3). In this chapter we will test them as splitting parameters, using the MDP-DT, which utilizes Decision Trees and is also optimal.

We are going to test the 10 metrics as splitting parameters in several setups. Firstly, we will use each one alone. Secondly, we will use them in groups, as they were grouped in chapter 4. Finally, we will use them all together. For each setup we will evaluate Tiramola's performance and therefore we will draw conclusions about how much each metric helped as a splitting parameter.

For doing such evaluation, we are going to keep the same policy, which is "*we want the cluster to have the smallest possible size, but always serve the incoming load*". Based on this policy, we are going to define the ideal performance of Tiramola for a sinusoidal load. Therefore, every time we do an experiment we will compare the current performance of Tiramola against the ideal one and introduce our measure of comparison which is the Mistake (5.2.3).

Having completed our evaluation of the splitting parameters and having defined the optimal ones, we go to the 4th and last phase of experiments. In that phase we test Tiramola with MDP-DT algorithm and all optimal parameters, general and splitting ones, by sending unpredictable load against the HBase cluster. This last phase brings new challenges like defining the level of randomness of an unpredictable load and the level of Tiramola's flexibility.

5.2 Experiments pt. 3: Sinusoidal load

5.2.1 Metrics as splitting parameters

As it is stated in chapter 4, the basic parameter that defines the environment of the Tiramola-agent is the number_of_VMs. The second most selected parameter is the next_load. These two parameters are used in Q and MDP algorithms and are selected by the user with their values defined. In this chapter we are going to use Decision Trees, by experimenting with the MDP-DT algorithm, which uses the splitting algorithm [6]. In MDP-DT we define some of the metrics as parameters and let the algorithm grow the Decision Tree starting from only few States. Each time a split happens, the algorithm decides which of the parameters is more suitable for splitting the current State into two new States and in which point.

Also, we can help the algorithm by defining an initial parameter with its values. In this case the algorithm knows from the beginning 1 Dimension of the State Space and starts with a Tree having a small number of States from the beginning. The work [6] defines the "small number of States" as 6 or something similar. In this work we choose to use as initial parameter the number_of_VMs by defining all its possible values: 3, 4, 5, 6, 7, 8, 9. So, the algorithm will fully know from the beginning the 1st Dimension of the State Space and start the Decision Tree with 7 States. We do this for several reasons.

- number_of_VMs is the most critical parameter that describes the cluster because the size of the cluster is what matters most for the Tiramola-agent.
- During the preliminary experiments with Decision Trees (MDP-DT), we noticed that the algorithm always preferred the number_of_VMs as the splitting parameter among the others and always found out the exact splitting points/values which were identical with the different sizes of the cluster.
- The MDP-DT spends many time-steps doing splits and growing the Decision Tree (State Space). The bigger the Tree, the more detailed the description of the environment within the Tiramola-agent acts. Obviously there is no need to deprive this knowledge and let the algorithm spending time-steps for doing splits considering number_of _VMs, given that we can define both the number_of_VMs as a parameter and its values/splitting points from the beginning.
- Another reason for this decision is that we always know the accurate possible sizes of the NoSQL-cluster (in our case, 3 - 9 VMs) and can easily define the number_of_VMs' values in the .json file where all the parameters are defined. On the other hand, it is not equally easy to know the possible values of all the other parameters!

5.2.2 The splitting algorithm

The splitting algorithm is involved in both models that have Decision Trees, the Q-DT and the MDP-DT. Taking into consideration the Tiramola's workflow, the splitting algorithm is part of the (g) step, and runs right after updating the values of the current State and Q-State. When the conditions are mature, it splits the current State by replacing it with a Decision Node and creates 2 new States and thus makes the Decision Tree to grow bigger.

From the beginning of the MDP-DT algorithm a vector of all States is created. As required for a Markov Decision Process, a list of Q-states is stored in each State, and each one of the Q-States corresponds to a possible Action the Tiramola-agent can do. Each Q-state holds the number of transitions and sum of rewards towards each State, along with the total number of times its corresponding Action has been taken. The Tiramola-agent knows its current State S_1 from the retrieved metrics M_1. It decides to take Action A, the NoSQL-cluster modifies its size and the clients run the load against it. During the load-time, the Monitoring module gathers the metrics M_2 and when the load-time is over, the Tiramola-agent obtains its Reward R. The value of the Reward is determined by the reward function, which is user-defined and usually depended by the value of one or more metrics of the M_2 set. Now, the Tiramola-agent has all the required information to update the Values of State S_1 and Q-Values of S_1's Q-States, according to the user-selected update algorithm. Except of these values, it also updates all the variables that define the number of transitions and other valuable statistics.

One of the previously mentioned as "valuable statistics" is the quartet $<M_1, A, M_2, R>$ that corresponds to metrics M_1 and M_2, the obtained Reward R and the selected Action A, as described in the previous paragraph. This quartet is stored in the State S_1's list that corresponds to S_2. When the updating ends, the algorithm checks for a possible split on State S_1, so it follows this workflow:

(i) It retrieves the current best Action of S_1, which is the Action that corresponds to the Q-State with the higher Q-Value of all S_1's Q-States.

(ii) It isolates the experiences $<M_1, A, M_2, R>$ where the Action happened and by using each quartet finds the State S_2 that corresponds to M_2 based on the current Decision Tree and calculates $q(m, a) = r + \gamma V(s')$, that is called "instantaneous Q-Value".

(iii) For each user-defined parameter p it calculates all the tuples $<m[p], q(m, a)>$ and sorts them based on the value of the parameter m[p].

(iv) For each two consecutive unequal values of the parameter $m_i[p]$ and $m_{i+1}[p]$ in that list, we consider splitting the state at their mid-point. For that purpose it runs a statistical test on the sets of instantaneous Q-values dividing them in two groups. The $q_- = \{q(m_k, a) \mid k <= i\}$ and the $q_+ = \{q(m_k, a) \mid k > i\}$. Each of the groups must have at least a number of values equal to the user-defined min_num_experiences. If not, splitting the State on the current point is aborted.

(v) If the condition in (iv) is fulfilled, the q_- and the q_+ groups are fed in a statistical test to check if they are statistically indifferent. The result of the statistical test is compared each time to the user-defined max_type_I_error. If the result is lower than max_type_I_error, the mid-point is a possible splitting point.

For each parameter we have many possible splitting points to check. For all these points that fulfil all the conditions the winner is the one with the lowest achieved result of the statistical test. This point becomes a possible splitting point. So, we have only one splitting point for each one of the parameters. If there are more than one parameters with a possible splitting point, we also chose the one with the lowest achieved result in the statistical test. This means that when the splitting algorithm is running, only one point can be a splitting point and each time we have only one split at most. As it is obvious, there is a tuple that describes each split: <splitting_parameter, splitting_point>.

5.2.2.1 Assumptions on the splitting algorithm

It is obvious from step (iii) that if the parameter has always constant value, no real sorting can be done. Also, in this case, step (iv) is practically aborted. The parameter has always the same value, so the algorithm cannot define any splitting point.

The splitting algorithm tries to correlate the values of a parameter with the level of success of an Action. If the parameter's values vary a lot under the same circumstances (load, size of NoSQL-cluster), as we noticed in experiments with constant load, or do not have any certain pattern when the load or the size of the NoSQL-cluster changes, as we noticed in the linear load experiments, the algorithm will not be able to be efficient. The instantaneous Q-Values will be correlated with parameter-values that explain practically nothing, because these parameters cannot describe reliably the effort of the NoSQL-cluster in different circumstances. In such cases the split won't be efficient and the resulting States won't be useful for the Tiramola-agent to be aware of the environment.

Our assumption is that metrics of the 2nd group will produce worse Decision Trees and thus worse State Spaces than the ones of the 1st group, if they are used as splitting parameters for the *-DT algorithms.

5.2.3 Experimental setup

In the previous work [6] there is a lot of effort on defining the most efficient algorithms and the best values for their parameters. We will take into consideration the previous work and we will choose only the best algorithms and parameters in order to focus on defining the more and less efficient parameters for the splitting algorithm. The selected algorithms and parameters in the following experiments are:

- The HBase-cluster could expand and contract from 3 VMs to 9 VMs (including master).
- The Tiramola-agent could decide among the Actions: add 1 VM, add 2 VMs, no action, remove 1 VM and remove 2 VMs.

epsilon	0.5
RL model	MDP-DT
Update Algorithm	Prioritized Sweeping
Update Algorithm error	0.1
Max Steps	100
Initial Q-Values	0
Discount γ	0.5
Splitting Algorithm	Q-value test (mid-point)
Split error (max_type_I_error)	0.005
Minimum number of experiences	2
Statistical Test	Mann-Whitney test

Table 5.1: *Selected parameters for MDP*

	master	**slave**	**client**
number of VMs	1	8	4
vcpu	4	2	1
RAM (GB)	16	4	2
storage (GB)	10	40	10

Table 5.2: *VM characteristics for experiments*

We loaded 3,000,000 records in the HBase and by defining the HBase-parameter hbase.hregion.max.filesize to 32 MB we managed to have about 650 regions. The HDFS replication factor was set to 2.

In each of the following experiments we run sinusoidal load from 1,000 to 19,000 reqs/sec. We let the Tiramola agent to train for 2,000 time-steps while the epsilon parameter is set to 0.5. This means that the agent in each time-step has 50% possibility to choose a random Action (exploration) and 50% to decide the optimal Action (exploitation). Then, it runs for 200 more steps exploiting the accumulated knowledge and choosing only the optimal Action. Each load-period needs 40 steps, so all the evaluation time-steps were equal to 5 load-periods. In the following tables we study Tiramola's behavior and efficiency during the first 2 load-periods of evaluation and during the last (5th) load-period of evaluation.

5.2.4 Introducing comparison measurement

Based on the policy "*we want the cluster to have the smallest possible size, but always serve the incoming load*", we define the ideal performance of Tiramola for a sinusoidal load. During every time step the cluster is stressed by a specific load and there is an optimal size of the HBase-cluster with which the Tiramola-agent obtains the biggest Reward. For each time-step in the experiments we compare the selected size of the cluster by Tiramola with the optimal one and find the difference. If Tiramola selects the size of the cluster to be X, but the optimal size is X-1 or X+1, we say that this is 1 **Mistake**. In this way we distinguish clearly Tiramola's performance in each experiment. By changing only the splitting parameter in the whole experiment setup, Tiramola's performance determines the effectiveness of each splitting parameter.

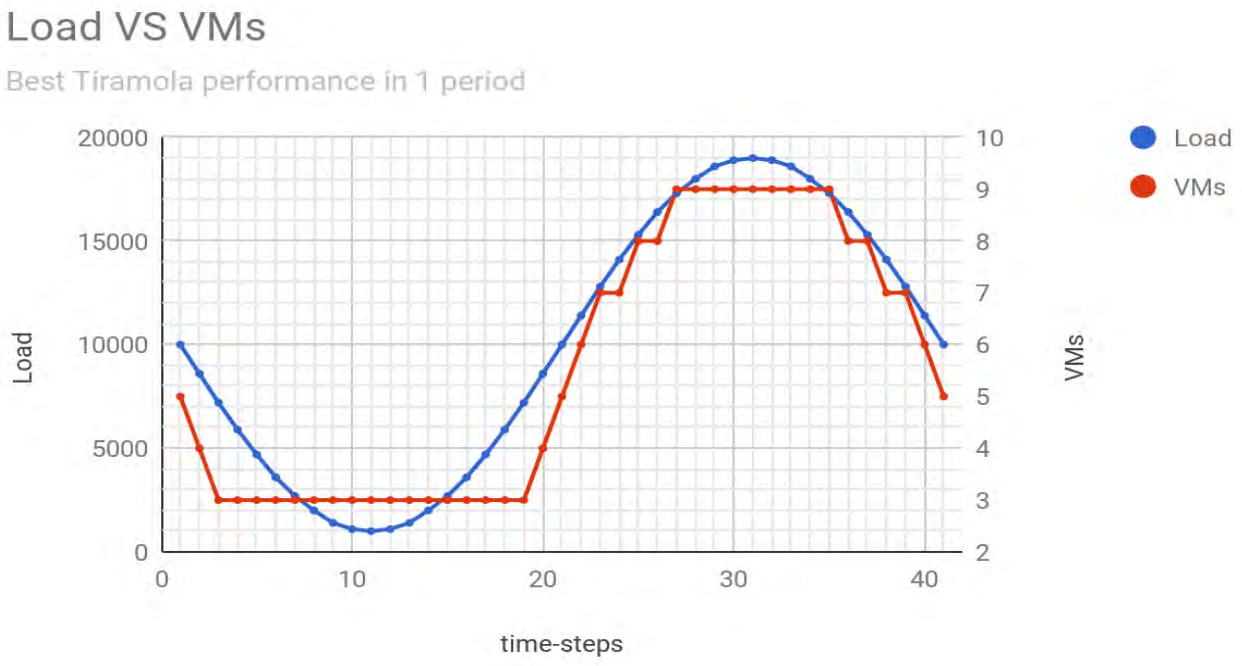

Figure 5.1: *Ideal Tiramola performance against sinusoidal load*

5.2.5. Performance of splitting parameters

We ran every experiment with number_of_vms being the default Dimension of the State Space. We use every metric as the only splitting parameter and run each experiment 10 times. Each time we run an experiment, we calculate the mistakes for the 1st, 2nd and 5th period during exploitation. In this set of experiments the whole training lasts 2000 time steps (50 sinusoidal periods), while the exploitation lasts 200 time steps (5 sinusoidal periods). We exclude the best and the worst performance based on the number of mistakes and calculate the average of mistakes of the remaining 8 experiments.

In the experiments where we use multiple metrics as splitting parameters, we expect Tiramola to split the State Space faster, so we are stricter and train Tiramola for 1000 time steps (25 sinus periods).

<table>
<tr><th></th><th>Parameters</th><th>AVG num of Mistakes
1st and 2nd period</th><th></th><th>AVG num of Mistakes
5th period</th><th></th></tr>
<tr><td colspan="4">2000 time-step-training</td><td></td></tr>
<tr><td rowspan="5">1st group</td><td>%_CPU_usage</td><td>36.3</td><td rowspan="5">1st group calculated average:
37.05</td><td>16.3</td><td rowspan="5">1st group calculated average:
16.9</td></tr>
<tr><td>net_usage</td><td>40.6</td><td>19.6</td></tr>
<tr><td>total_throughput</td><td>37.3</td><td>18.6</td></tr>
<tr><td>total_latency</td><td>36.6</td><td>15</td></tr>
<tr><td>load_one</td><td>38</td><td>15</td></tr>
<tr><td rowspan="5">2nd group</td><td>io_reqs</td><td>85.3</td><td rowspan="5">2nd group calculated average:
129.5</td><td>78.3</td><td rowspan="5">1st group calculated average:
73.26</td></tr>
<tr><td>cpu_wio</td><td>119</td><td>58</td></tr>
<tr><td>%_free_RAM</td><td>154</td><td>73</td></tr>
<tr><td>disk_free</td><td>197.6</td><td>95</td></tr>
<tr><td>%_cached_RAM</td><td>91.6</td><td>62</td></tr>
<tr><td colspan="2">next_load</td><td>31.6</td><td></td><td>10.3</td><td></td></tr>
<tr><td colspan="4">1000 time-step-training</td><td></td></tr>
<tr><td colspan="2">1st group + next_load</td><td>33.25</td><td></td><td>15</td><td></td></tr>
<tr><td colspan="2">2nd group</td><td>82.75</td><td></td><td>28.75</td><td></td></tr>
<tr><td colspan="2">All</td><td>96</td><td></td><td>24.75</td><td></td></tr>
</table>

Table 5.3: *Tiramola performance for one or more splitting parameters*

When Tiramola is using a splitting parameter of the 1st group of parameters the average Mistakes for 2 periods right after training is approximately 37. The following chart is an example of how Tiramola's performance looks like using total_throughput as a parameter in a similar case.

Figure 5.2: *Tiramola performace with total_thgoughput as splitting parameter*

When Tiramola is using a splitting parameter of the 2nd group, the average Mistakes for 2 periods right after training is approximately 130. The following chart is an example of how Tiramola's performance looks like using cpu_wio as a parameter in a similar case.

Figure 5.3: *Tiramola performace with cpu_wio as splitting parameter*

When Tiramola is using a splitting parameter of the 1st group, the average Mistakes during the 5th period after the training is approximately 16. The following chart is an example of how Tiramola's performance looks like using total_latency as a parameter in a similar case.

Figure 5.4: *Tiramola performace with total_latency as splitting parameter*

When Tiramola is using a splitting parameter of the 2nd group, the average Mistakes during the 5th period after the training is approximately 73. The following chart is an example of how Tiramola's performance looks like using %_free_RAM as a parameter in a similar case.

Figure 5.5: *Tiramola performace with %_free_RAM as splitting parameter*

When Tiramola is using next_load, %_CPU_usage, network_usage, total_throughput, total_latency and load_one (the whole 1st group) as splitting parameters, the average Mistakes during the 1st exploitation sinus period is around 33, while during the 5th period is 15. The following chart shows a similar performance.

Figure 5.6: *Tiramola performace with all 1st group as splitting parameters*

When Tiramola is using io_reqs, cpu_wio, %_free_RAM, disk_free and %_cached_RAM (the whole 2nd group) as splitting parameters, the average Mistakes during the 1st exploitation sinus period is approximately 80, while during the 5th is 29. The following chart show a similar performance.

Figure 5.7: *Tiramola performace with all 2nd group as splitting parameters*

5.2.6 Conclusions on splitting parameters' performance

It is obvious that the parameters of the 1st group have better performance than the ones of the 2nd group when used separately.

The more Tiramola is going further from training period the more it improves its performance when it uses parameters from the 1st group. On the other hand this happens less often when it uses the ones of the second, and when it does, the improvement on its performance is not that great.

When using parameters of the 1st group altogether, Tiramola's performance is better than using those of the 2nd altogether.

When we use all 11 parameters the performance is worse than using only the ones of the 1st group. That means that parameters of the 2nd group are not only worse when used alone, but they also harm the whole performance.

Using more parameters is not necessarily a smart choice. More Dimensions in State-Spaces doesn't imply better performance.

5.3 Experiments pt. 4: Unpredictable Load

After defining the best parameters and algorithms of Tiramola, we are going to run unpredictable load against the HBase-cluster and we will study Tiramola's performance. Tiramola's configuration is the same as in the previous stages and the chosen parameters are those of the 1st group. Never before the Tiramola has been tested in such a load, so we are going to do a lot of preliminary tests in order to define the level of unpredictability of the load, the range of the its values and when Tiramola is able to react by taking into account the frequency in load's transitions from very low values to very high ones. Summing up, we are going to study and define a fair unpredictable load according to Tiramola's skills and then come to a conclusion about Tiramola's speed of reaction.

We run 4 different types of unpredictable load against the HBase-cluster.

unpredictable_load1: In every time step, each of the 7 different loads is randomly chosen and runs against the cluster.

unpredictable_load2: The same load stresses the cluster for 2 successive time steps. After that, the load's value is a new random one.

unpredictable_load3: The same load stresses the cluster for 3 successive time steps. After that, the load's value is a new random one.

unpredictable_load4: The same load stresses the cluster for 4 successive time steps. After that, the load's value is a new random one.

We run every load in 4 different sessions changing only the number of training time-steps. Tiramola has different number of training time steps: 1000, 2000, 4000 and 8000. During each experiment Tiramola runs for 100 more time steps choosing the optimal Action. We present its performance in the next charts.

5.3.1. HBase cluster stressed under 1st type unpredictable load

Figure 5.8: *Tiramola performance: 1st type of Unpredictable load. 1000 t.s. training*

Figure 5.9: *Tiramola performance: 1st type of Unpredictable load. 2000 t.s. training*

Figure 5.10: *Tiramola performance: 1st type of Unpredictable load. 4000 t.s. training*

Figure 5.11: *Tiramola performance: 1st type of Unpredictable load. 8000 t.s. training*

5.3.2 HBase cluster stressed under 2nd type unpredictable load

Figure 5.12: *Tiramola performance: 2nd type of Unpredictable load. 1000 t.s. training*

Figure 5.13: *Tiramola performance: 2nd type of Unpredictable load. 2000 t.s. training*

Figure 5.14: *Tiramola performance: 2nd type of Unpredictable load. 4000 t.s. training*

Figure 5.15: *Tiramola performance: 2nd type of Unpredictable load. 8000 t.s. training*

5.3.3 HBase cluster stressed under 3rd type unpredictable load

Figure 5.16: *Tiramola performance: 3rd type of Unpredictable load. 1000 t.s. training*

Figure 5.17: *Tiramola performance: 3rd type of Unpredictable load. 2000 t.s. training*

Figure 5.18: *Tiramola performance: 3rd type of Unpredictable load. 4000 t.s. training*

Figure 5.19: *Tiramola performance: 3rd type of Unpredictable load. 8000 t.s. training*

5.3.4 HBase cluster stressed under 4th type unpredictable load

Figure 5.20: *Tiramola performance: 4th type of Unpredictable load. 1000 t.s. training*

Figure 5.21: *Tiramola performance: 4th type of Unpredictable load. 2000 t.s. training*

Figure 5.22: *Tiramola performance: 4th type of Unpredictable load. 4000 t.s. training*

Figure 5.23: *Tiramola performance: 4th type of Unpredictable load. 8000 t.s. training*

5.3.5 Conclusions on Tiramola's performance under unpredictable load

Against 1st type of unpredictable load

Tiramola is completely incapable of following such load. Only after a big number of training time steps (only at 8000) it seems to react a bit, but the changes in the cluster's size seem hopeless. However, we should remind that based on the reward function we defined, Tiramola must choose the smallest cluster size that serves the incoming load (the less cost). Considering that fact, Tiramola cannot practically follow such an unpredictable load and chooses to keep the cluster's size in an average value and chooses 6 VMs (3 VMs minimum, 9 VMs maximum). In this way, it achieves a balance between serving the loads and keeping the cost low.

Against 2nd type of unpredictable load

Each load is random, but changes every 2 time steps. Tiramola now has one more time step to modify the cluster's size, but fails to adequately follow the load. By increasing the training time-steps, Tiramola seems to be more stable, but still performing poorly.

Against 3rd type of unpredictable load

During the 3rd type of unpredictable load, Tiramola has 2 more time-steps to recognize the load. Under this load Tiramola starts performing well, especially when the training lasts 4000 time-steps or more.

Against 4th type of unpredictable load

During the 4th type of unpredictable load, Tiramola has more opportunities to react and adjust the cluster's size to the incoming load. It is not only performing well after 4000 time-steps of training or more, but also achieves a slightly good performance even after 2000 time steps of training.

5.3.6 Extending Tiramola's flexibility

Every time the load's value changes the possible next value can be any of 7 predefined values. In the 2 worst cases for Tiramola the load goes from the lowest value to the highest one or the opposite. In such cases it is possible that the cluster has the smallest available size (3 VMs). In the case of the total increase of the load Tiramola needs to expand the cluster to its biggest available size (9 VMs). During these experiments we kept the previous availability of Actions for Tiramola, meaning that the biggest expansion or contraction of the cluster is by 2 VMs. So, Tiramola needs 4 time steps to expand the cluster by 6 VMs (2 + 2 + 2), considering that during the first time

step Tiramola has no way of knowing how much the load will change. In conclusion, if Tiramola is able to choose only such Actions, then it is fair to study its performance only against unpredictable loads of the 5th type. Tiramola's available flexibility must be proportionate to the changes of the load. In the next experiments that will follow we will enable more Actions for the Tiramola, and we will study its performance under such unpredictable loads.

As the previous experiments showed, it is unfair to have extreme changes in the load, i.e. from 6000 reqs/sec to 14000 reqs/sec, that can be served by 3 and 7 VMs respectively, and Tiramola's maximum flexibility being only 2 VMs (plus or minus). So, we increased Tiramola's flexibility by leaving all the other features as they were. Now Tiramola can add or remove 1, 2, 3, 4, 5 and 6 VMs. Even so, Tiramola and no other system can be able to predict a load that is completely unpredictable, meaning that Tiramola can never follow adequately a load that is different in every time-step, previously presented as "unpredictable load of 1st type". So, we are going to test Tiramola by stressing the HBase-cluster under loads of 2nd and 3rd type. Also, we will randomly select a specific sequence of both loads for 100 time-steps that will be the evaluation sequences. We will going to define the ideal reaction of Tiramola for these 100 steps in each case, thus being able to accurately define Tiramola's performance.

5.3.6.1 Tiramola against unpredictable load of 2nd type

As previously mentioned, the load changes randomly every 2 time steps. During the 1st time-step Tiramola is unable to predict the load, so we stress the cluster with the same load for one more step giving a realistic opportunity to Tiramola to follow it. Such load is stressing the cluster for the whole training session, but we have a specific sequence of this type of 100 loads in every experiment during the evaluation period. In the next chart we can see this sequence of 100 loads and the ideal reaction of Tiramola, if it was able to fully predict the load.

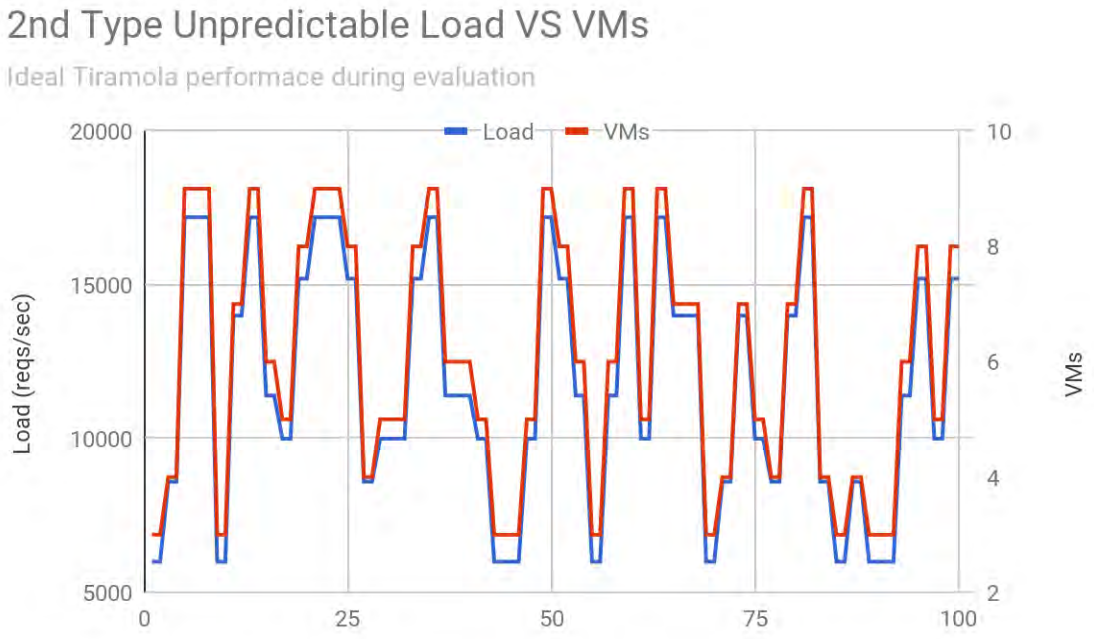

Figure 5.24: *Ideal Tiramola performance against 2nd type of unpredictable load*

We evaluate Tiramola after 1000, 2000, 4000 and 8000 training steps. After every training session the same sequence of loads for 100 time-steps happens during which we study Tiramola's behavior. Stressing the cluster under the same sequence of loads we compare Tiramola's reaction with the ideal one and count the Mistakes, our comparison measurement, as defined in previous experiments with the sinusoidal load (if Tiramola chooses 4 VMs, but the ideal choice is 3 or 5 VMs, this is 1 mistake and so on).

Every experiment is conducted 10 times and each time we study Tiramola's performance by counting mistakes. Then, we neglect the best and the worst performance and calculate the average number of mistakes based on the other 8 times we conducted the experiment. Except of studying Tiramola's performance for the whole 100 time steps we will pay more attention on Tiramola's performance during each 2nd time-step of the evaluation sequence. During this time-step Tiramola has its realistic opportunity to react.

Unpredictable Load of 2nd type	**1000 training time-steps**	**2000 training time-steps**	**4000 training time-steps**	**8000 training time-steps**
AVG mistakes at 100 evaluation t.s.	167.4	155.3	136.6	131.1
AVG mistakes every 2 eval. t.s.	66.3	52.1	39.6	35.1

Table 5.4: *Tiramola performace against 2nd type of unp. load. Standard evaluation load*

In the next 4 charts we see how Tiramola's performance looks like in similar cases as depicted in the table above:

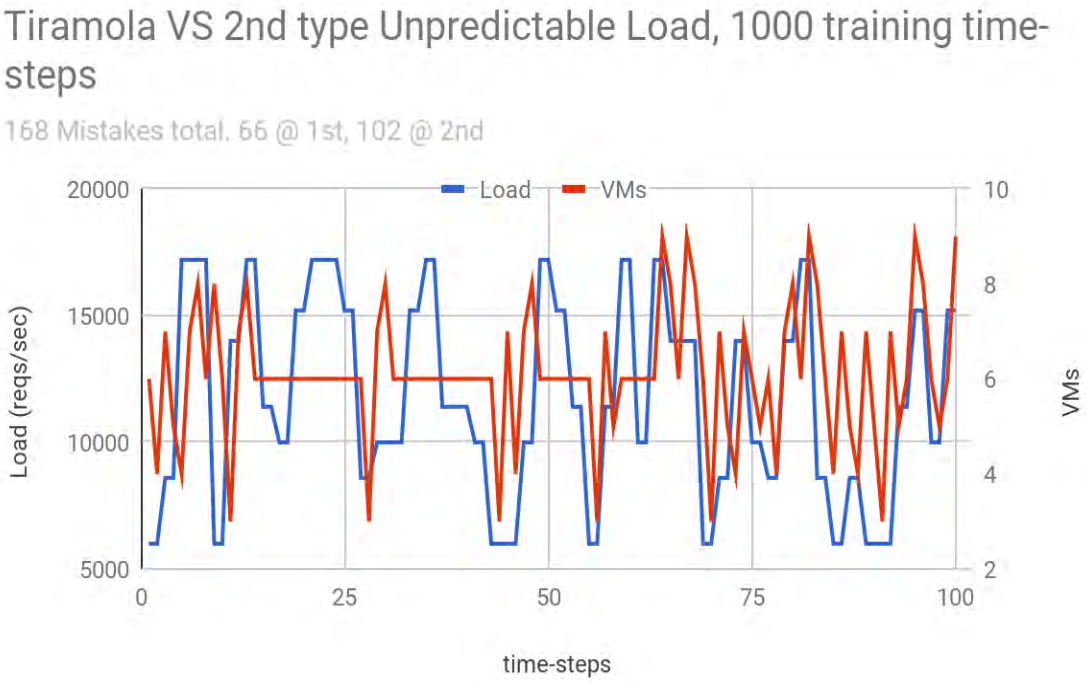

Figure 5.25: *Tiramola perf.: Standard evaluation 2nd type of unpr. load, 1000 train t.s.*

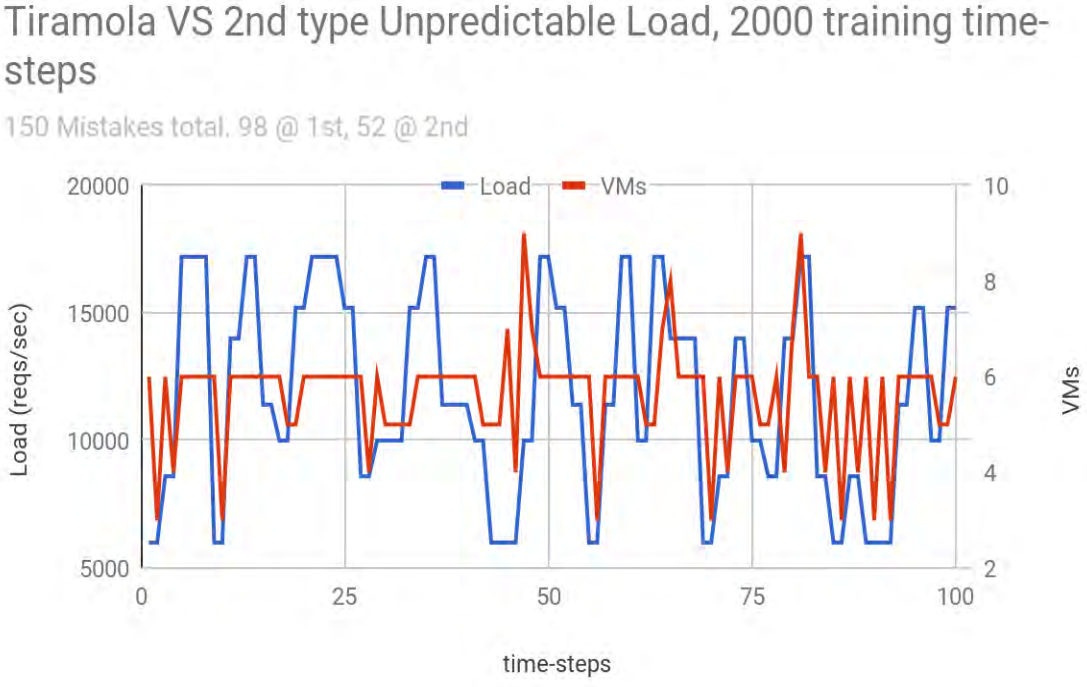

Figure 5.26: *Tiramola perf.: Standard evaluation 2nd type of unpr. load, 2000 train t.s.*

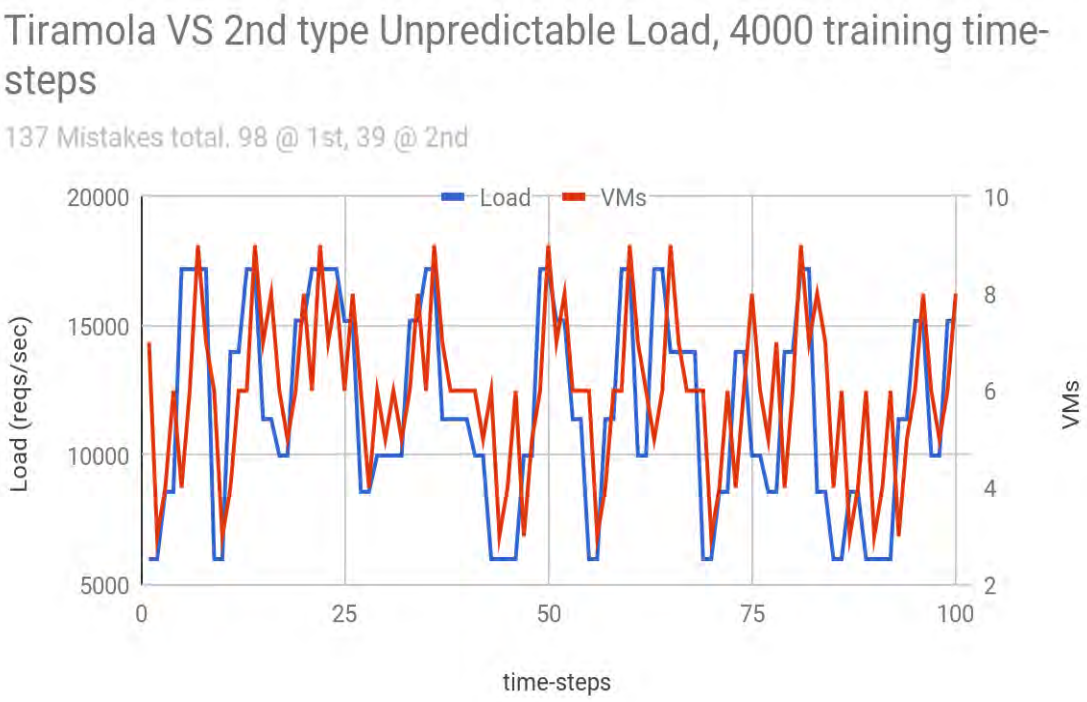

Figure 5.27: *Tiramola perf.: Standard evaluation 2nd type of unpr. load, 4000 train t.s.*

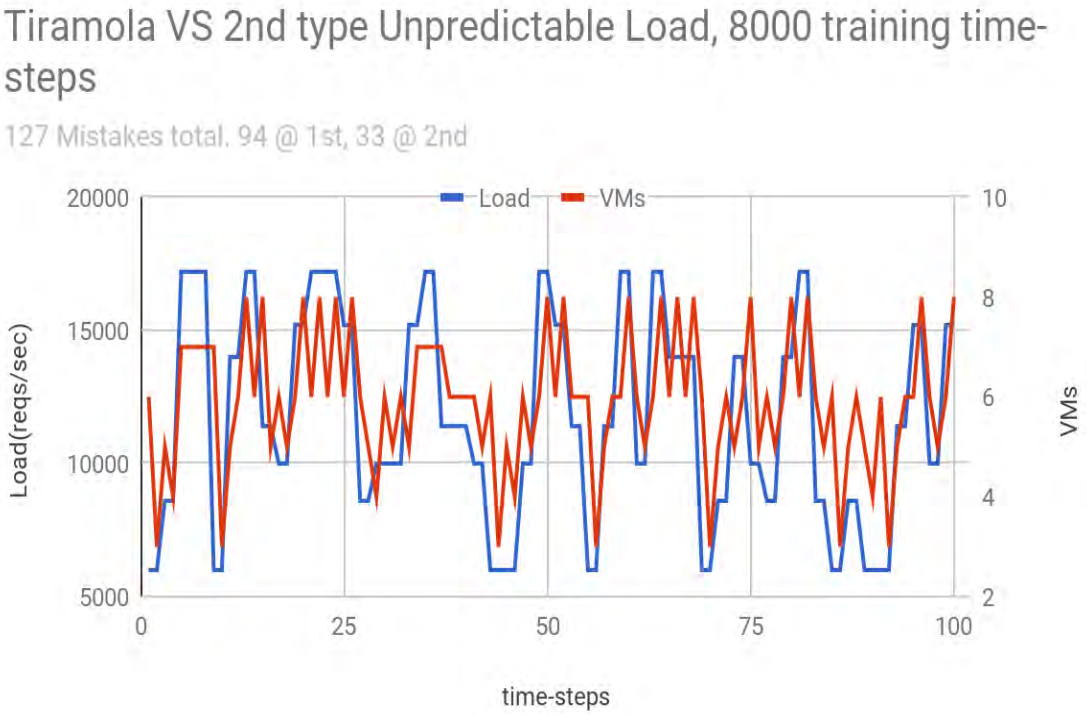

Figure 5.28: *Tiramola perf.: Standard evaluation 2nd type of unpr. load, 8000 train t.s.*

5.3.6.2 Tiramola against unpredictable load of 3rd type

As previously mentioned, the load changes randomly every 3 time steps. During the 1st time-step Tiramola is unable to predict the load, so we stress the cluster with the same load for two more steps. Now Tiramola has 2 opportunities to follow the load and adapt the size of the cluster. Such load is stressing the cluster for the whole training session, but during the evaluation we stress the cluster under the same sequence of 100 loads in every experiment. In the next chart we can see this sequence of 100 loads and the ideal reaction of Tiramola, if it was able to fully predict the load.

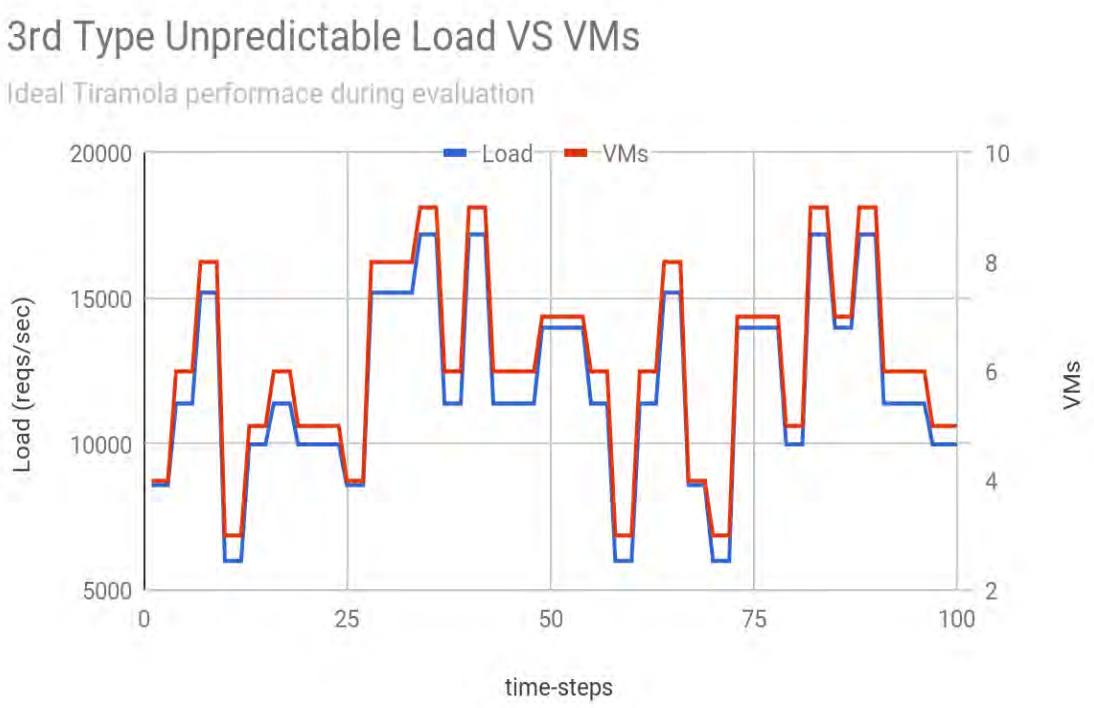

Figure 5.29: *Ideal Tiramola performance against 3rd type of unpredictable load*

We evaluate Tiramola after 1000, 2000, 4000 and 8000 training steps. After every training session the same sequence of loads for 100 time-steps happens during which we study Tiramola's behavior.

Every experiment is conducted 10 times and each time we study Tiramola's performance by counting mistakes. Then, we neglect the best and the worst performance and calculate the average number of mistakes based on the other 8 times we conducted the experiment. Except of studying Tiramola's performance for the whole 100 time steps we will pay more attention on Tiramola's performance during the 2nd and 3rd time-step every 3 time-steps of the evaluation sequence. During these 2 time-steps Tiramola has 2 realistic opportunities to react.

Unpredictable Load of 3rd type	1000 training time-steps	2000 training time-steps	4000 training time-steps	8000 training time-steps
AVG mistakes 100 evaluation time-steps	126	105.1	94.2	89.8
AVG mistakes on 2nd and 3rd eval t.s.	64.3	46	33.7	27

In the next 4 charts we see how Tiramola's performance looks like in similar cases as depicted in the table above:

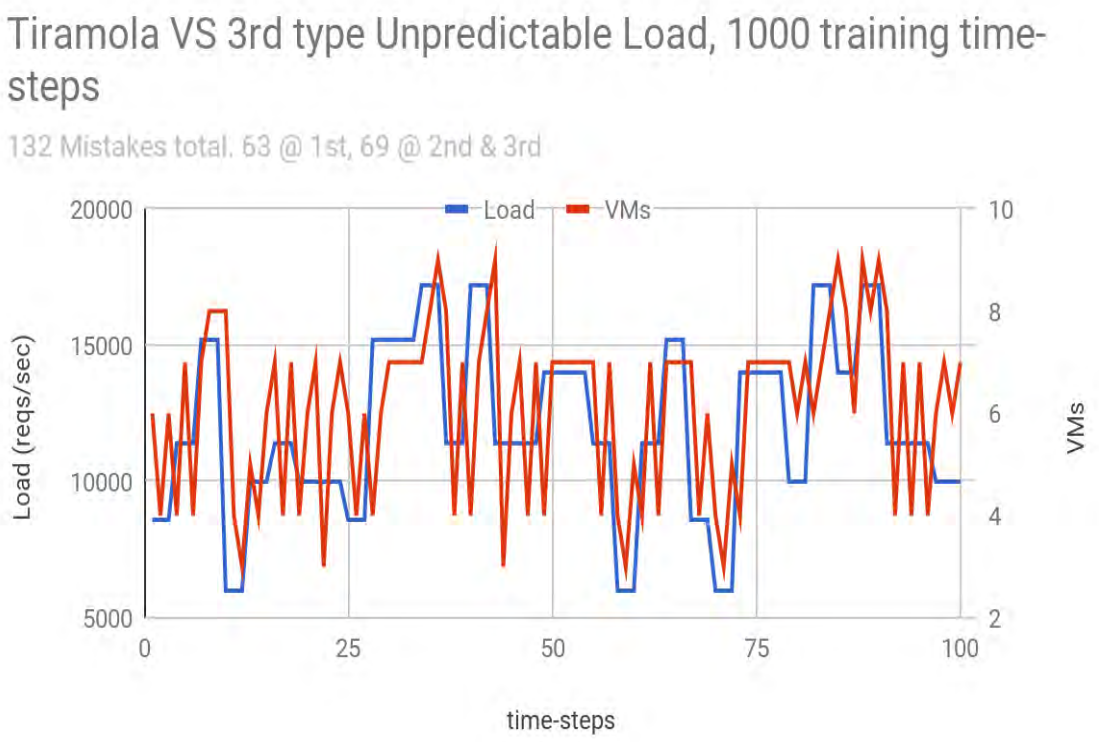

Figure 5.30: *Tiramola perf.: Standard evaluation 3rd type of unpr. load, 1000 train t.s.*

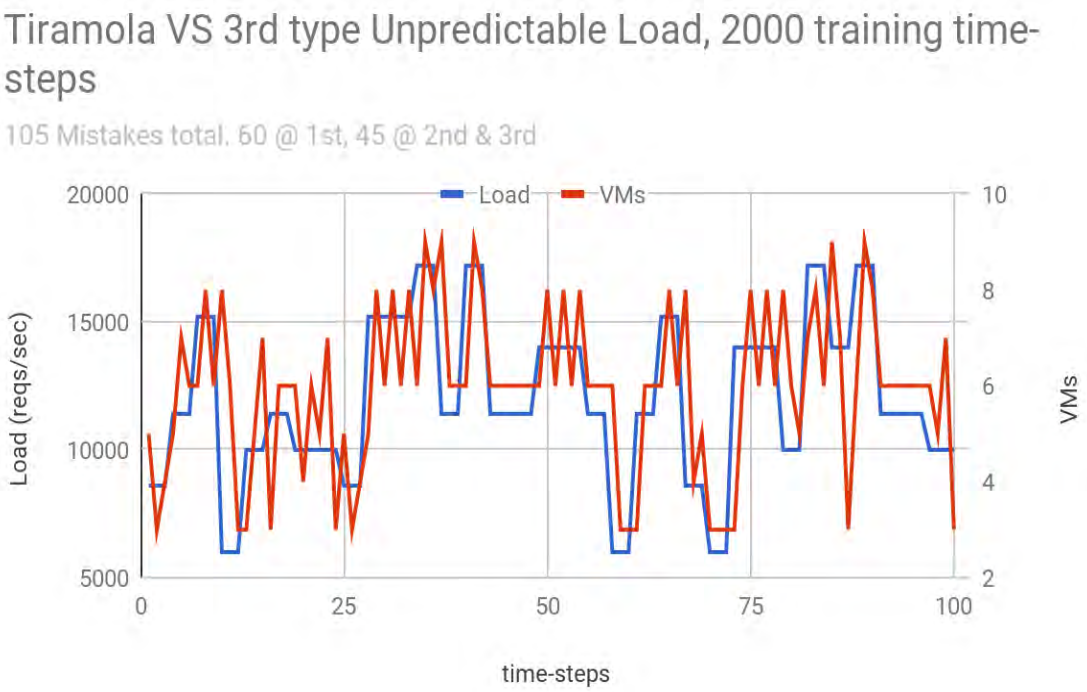

Figure 5.31: *Tiramola perf.: Standard evaluation 3rd type of unpr. load, 2000 train t.s.*

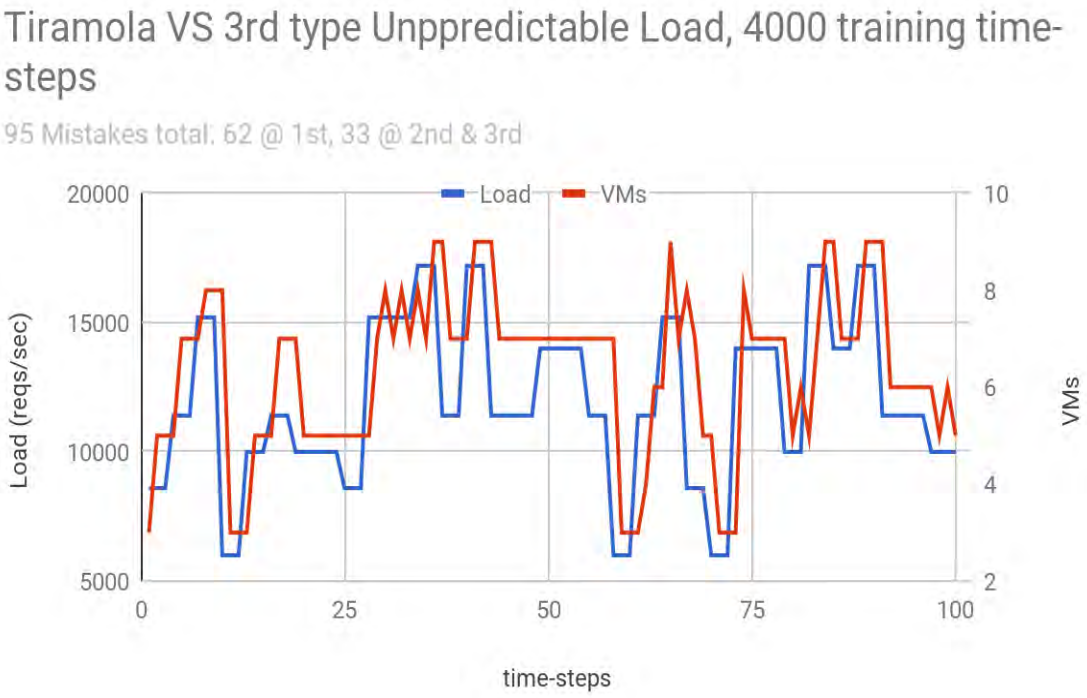

Figure 5.32: *Tiramola perf.: Standard evaluation 3rd type of unpr. load, 4000 train t.s.*

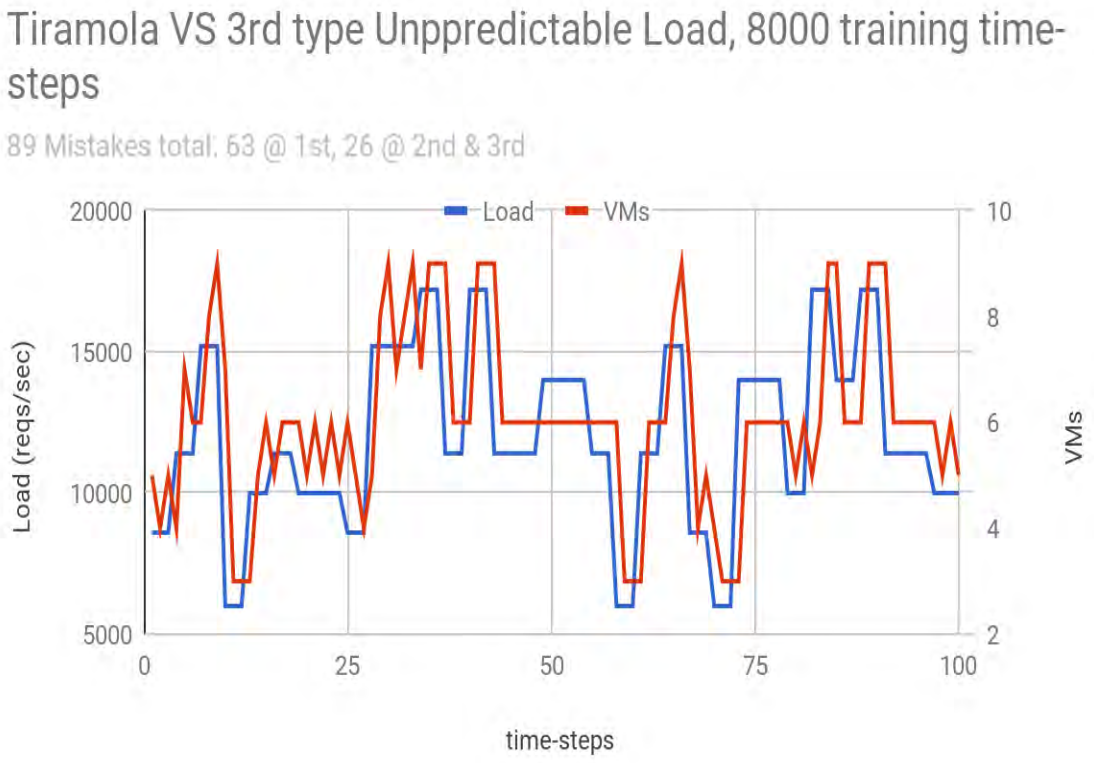

Figure 5.33: *Tiramola perf.: Standard evaluation 3rd type of unpr. load, 8000 train t.s.*

5.3.7 Conclusions on extended Tiramola performance against unpredictable load

By making Tiramola's flexibility proportionate to the volume of load-changes, Tiramola able to have much better performance under more demanding loads. Under unpredictable load of the 2nd type Tiramola performs sufficiently after 4000 training time-steps. In the previous experiments, having less flexibility, Tiramola's behavior was poor even after 8000 training time-steps. In the case of 3rd type of unpredictable load, Tiramola is performing sufficiently after 2000 training time-steps, while in the previous experiments it needed at least 4000 training time-steps to follow the load. So, if we let Tiramola to be flexible enough and train it sufficiently enough, Tiramola can have a good performance under unpredictable loads.

Chapter 6

EPILOGUE

In this work we used the last version of Tiramola which implements Markov Decision Process with Decision Trees that use the HBase cluster's metrics as splitting parameters and create a multi-dimension State Space based on them. We evaluated the performance of Tiramola when an HBase cluster is stressed under unpredictable load and Tiramola is trying to contract or expand the cluster under the policy "*keep the cluster to the smallest possible size, but always serve the incoming load*".

For doing such challenging task, we decided that it is of high importance to tune Tiramola to its optimal condition. All the MDP related parameters (epsilon, update algorithm, discount γ etc. [see 5.2.3]) were already evaluated and we already knew the optimal values, so we focused on the splitting parameters. Tiramola's Decision Tree uses the NoSQL cluster's metrics, while cluster is stressed, as splitting parameters to create new States, so these metrics are also becoming the Dimensions of the State Space. All things considered, studying the behavior of these metrics in depth was very important.

6.1 Conclusions

In the 4th chapter we present several experiments where we don't use Tiramola, but we send two different kinds of loads against an HBase cluster, while keeping the cluster's size constant. This allows us to study the behavior of 10 metrics directly and 21 metrics indirectly. Given that the total amount of useful metrics is 40 and we cover CPU, memory, disk and network usage we consider that our study on cluster's metrics is complete.

During the 1st phase of experiments we run linear increasing load and realize that we can divide the metrics into two groups based on the behavior. The ones that have a consistent behavior while the load is increasing and the ones that have unpredictable behavior. Also, we define the critical load for each cluster size, which is the highest load that the HBase cluster serves the requests for each size.

During the 2nd phase of experiments we run several loads under the critical load for each size of the HBase cluster (3, 4, 5, 6, 7, 8 and 9 VMs, master included). Doing so, we managed to present some basic analytics about the metrics that helped us understand deeper the behavior of metrics.

Based on these two phases of experiments and thinking that the cluster's metrics will be used as splitting parameters and Dimensions of the State Space, we conclude that:

- The ones of the 2nd group (4.2.1) have high variance when the cluster's size is constant and is stressed under the same load multiple times and/or their values do not differ when the load or the cluster's size is changing. Thinking of them as splitting parameters and/or Dimensions, we can make the assumption that they will not be efficient. Both characteristics make them inappropriate for creating a State Space that will describe the environment for the Tiramola agent reliably. Also, when Tiramola is using any of them as splitting parameters cannot correlate reliably under which circumstances a specific State will bring positive or negative Reward.
- On the other hand, the metrics of the 1st group (4.2.1) have the opposite behavior and thus, they can be used as Dimensions of a State Space that reliably describes the environment for the Tiramola agent. Also, being consistent makes them reliable as splitting parameters.

In the 5th chapter we use Tiramola in two phases of experiments.

During the 3rd phase we confirm our assumptions about which metrics can be used as splitting parameters in Decision Trees and as Dimensions for the State Space. To accomplish that, we run sinusoidal load against the HBase cluster and do multiple experiments using each metric as a splitting parameter. In order to do the evaluation we introduce the Mistake, which is a comparison measurement for evaluating Tiramola's performance that can be used when a NoSQL cluster is stressed under a standard load.

During the 4th phase we used Tiramola optimally, while the HBase cluster was stressed under unpredictable load. In this part of experiments we manage to:

- Define different level of randomness for unpredictable loads (4 different types). Never before such loads were used in relevant works, so this was a challenging task.
- Define when Tiramola is performing acceptably: against what type of unpredictable loads and after how many training time steps.
- Go further by extending Tiramola's flexibility about contracting or extending the cluster. Doing so, we made the encounter (Tiramola VS unpredictable load) to be fairer and saw that Tiramola can perform better against unpredictable loads that have high randomness, while Tiramola is less trained.

Bibliography

[1] Konstantinou I., Angelou E., Boumpouka C., Tsoumakos D. and Koziris N., "On the elasticity of nosql databases over cloud management platforms", in Proceedings of the 20th ACM international conference on Information and knowledge management, pp. 2385–2388, ACM, 2011.

[2] Angelou E., Papailiou N., Konstantinou I., Tsoumakos D. and Koziris N., "Automatic scaling of selective SPARQL joins using the TIRAMOLA system", in Proceedings of the 4th International Workshop on Semantic Web Information Management, p.1, ACM, 2012.

[3] Konstantinou I., Angelou E., Tsoumakos D., Boumpouka C., Koziris N. and Sioutas S., "Tiramola: elastic nosql provisioning through a cloud management platform", in Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp.725–728, ACM, 2012.

[4] Tsoumakos D., Konstantinou I., Boumpouka C., Sioutas S. and Koziris N., "Automated, elastic resource provisioning for nosql clusters using tiramola", in The 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp.34 – 41, IEEE, 2013

[5] Kassela E., Boumpouka C., Konstantinou I. and Koziris N, "Automated workload-aware elasticity of NoSQL clusters in the cloud", in 2014 IEEE International Conference on Big Data (Big Data), pp. 195–200, IEEE, 2014

[6] Lolos K., Konstantinou I., Kantere V. and Koziris N., "Adaptive State Space Partitioning of Markov Decision Processes for Elastic Resource Management" in 2017 IEEE 33rd International Conference on Data Engineering, pp.191 – 194, IEEE, 2017.

[7] Lolos K., Konstantinou I., Kantere V. and Koziris N., "Elastic Management of Cloud Applications using Adaptive Reinforcement Learning", in proceedings of the 2017 IEEE International Conference on Big Data (BigData 2017), Boston, MA, USA, December 11-14 2017.

[8] OpenStack official site (2019), Retrieved from: https://www.openstack.org/

[9] White T., Hadoop: The Definitive Guide, 4th Edition. O'Reilly Media, Inc, CA (2015)

[10] Hadoop official site (2019), Retrieved from: https://hadoop.apache.org/

[11] Ghemawat S., Gobioff H. and Leung S. T., "The Google file system", in ACM SIGOPS operating systems review, vol. 37, pp. 29–43, ACM, 2003.

[12] George L., HBase: The Definitive Guide, 2nd Edition O'Reilly Media, Inc, CA (2015)

[13] HBase official site (2019), Retrieved from: https://hbase.apache.org/

[14] Chang F., Dean J., Ghemawat S., Hsieh W. C., Wallach D. A., Burrows M., Chandra T., Fikes A., and Gruber R. E., "Bigtable: A Distributed Storage System for Structured Data," in OSDI, 2006.

[15] Cooper B. F., Silberstein A., Tam E., Ramakrishnan R. and Sears R., "Benchmarking Cloud Serving Systems with YCSB", in ACM SOCC, 2010.

[16] Massie M., Li B., Nicholes B. and Vuksan V., Monitoring with Ganglia, 1st Edition O'Reilly Media, Inc, CA (2013).

[17] Ganglia official site (2019), Retrieved from: http://ganglia.sourceforge.net/

[18] Pyeatt L. D, Howe A. E. et al., "Decision tree function approximation in reinforcement learning", in Proceedings of the third international symposium on adaptive systems: evolutionary computation and probabilistic graphical models, vol.1, p.2, 2001.

[19] Sutton R. S. and Barto A. G., Reinforcement Learning: An Introduction, 2nd Edition The MIT Press Cambridge, MA (2017).

[20] Puterman M. L., Markov Decision Processes: Discrete Stochastic Dynamic Programming, 1st Edition John Wiley & Sons, Inc., New York, NY (1994).