University of London
Imperial College of Science, Technology and Medicine
Department of Computing

# Performance Modelling and Optimisation of NoSQL Database Systems

Salvatore Dipietro

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, October 2019

# Declaration of originality

I declare that this thesis was composed by myself, and that the work it presents is my own, except where otherwise stated.

# Copyright declaration

# Abstract

Over the last decade, the use of mathematical models and tools to describe and analyse computer applications have grown considerably, for example to automate management in the cloud. Modelling techniques help performance engineers to analyse the behaviour of the system under certain simplifying assumptions and predict its performance, often without running experiments on the real system. However, modern computer applications such as distributed applications can be challenging to describe using models and, even then, their analysis can be technically non-trivial also due to the number of resources involved and their interactions.

In this thesis, we consider modelling and optimisation of distributed NoSQL databases, focussing in particular on Apache Cassandra. NoSQL databases have attracted large interest in recent years thanks to their high availability, scalability, flexibility and low latency. Nonetheless, concrete implementations such as Cassandra are challenging to analyse since requests interact in complex ways with the nodes that form the database ring. We address the underpinning modelling and management challenges as follows.

We first propose a novel queueing network model for Cassandra to support database resource provisioning exercises. The model defines explicitly key configuration parameters of Cassandra such as consistency levels and replication factor, allowing engineers to compare alternative system setups. The experimental results are conducted using different architectures and hardware resources, achieving good predictive accuracy across different loads and consistency levels. In addition, we also present a case study where the model is used to perform capacity planning activities and to compare possible alternative consistency level definition strategies.

A second contribution focuses on management, where we introduce PAX, a partition-aware elastic resource management system for Apache Cassandra. PAX allows engineers to adapt NoSQL database resources to reduce operational costs without compromising Service-Level Objectives (SLOs). Using a low-overhead query sampling and knowledge of the data-partitioning across the nodes, PAX automatically adapts capacity in Cassandra clusters looking for the configuration that is able to achieve the best performance. We analyse the system using a reactive and a proactive implementation of PAX and compare their performance against different workloads with varying intensities and item popularity distributions, finding that in particular the proactive version of PAX significantly reduces SLO violations.

We also present a new estimation algorithm to instantiate performance models based on empirical measurements, called State Divergence (SD). Frequently, service demand estimation for real-world systems is calculated in testing environments that can have different characteristics compared to the production ones, leading to inaccurate performance predictions. SD offers a novel approach to demand estimation that has a minimal impact on the application and makes it suitable for application also in production environments. Differently from existing inference algorithms, SD seeks to minimise the divergence between marginal state probability of the real and analysed model to produce accurate demand estimates that reflect not only the performance metrics, but also the likelihood that the system is in an given state. We validate the SD estimation algorithm through several randomly generated models and by means of a real case study conducted on Apache Cassandra. The results show that SD infers with a low error the demands of the system under study and predicts with accuracy its performance, allowing to parameterise performance models with ease and higher fidelity than with existing methods.

# Acknowledgements

First of all, I would sincerely like to thank my academic supervisor Giuliano Casale for his continuous advice, encouragement and guidance throughout the last four years. He helped me to develop a critical thinking as well as a rigorous methodology through the development of this thesis.

Second, I would like to express my gratitude to the Engineering and Physical Sciences Research Council (EPSRC) to fund my HiPEDS studentship. In particular, I would like also to thanks all the lecturers, staff and students of my cohort that made these years a beautiful adventure.

Most importantly, I would like to thank my family for all the support I received to achieve this goal. During bad days, good days, milestones moments, difficulties; when you are ready to give up and just leave everything behind. You are always there with the right word, the right advice or even just being there, because I know I can count on you and you are my greatest supporters. Thank you.

To Guinevere and William, I dedicate my entire work. For all your life to be full filled of happiness, achievements and to always pursue your dreams and wishes. Growing is hard sometimes but with a strong will and positiveness everything is possible. I promise I will always be there to support you and guide you every step of the way.

“ Imagination is more important than knowledge.
For knowledge is limited, whereas imagination
embraces the entire world, stimulating progress,
giving birth to evolution. ”

*Albert Einstein*(1929)

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation and Objectives

Software performance requirements are usually described in terms of Quality-of-Service (QoS) levels expected from a system. Common QoS metrics include system availability, reliability, fault tolerance and performance metrics such as throughputs, resource utilisation, and response times. Quality-of-service levels for a software product are normally agreed between the service provider and the user in a contract called a Service Level Agreement (SLA) [RPS09]. QoS violations with respect to the parameters specified in the SLA contract usually cause a financial penalty for the service provider because such violations can impact severely the end-user business. Therefore, it is essential for service suppliers to guarantee that their services do not violate the SLA or, at least, to attempt to minimise the number of occurrences in which this happens in production. However, it is not possible in general to test a system under every possible workload and to address this limitation performance modelling techniques may be used to identify workloads that are potentially problematic for SLA compliance.

Over the last few decades, a wide range of mathematical techniques and tools have been developed to model and analyse QoS in computer software applications and infrastructures. These methods have helped in many instances performance engineers to analyse and meet software requirements using systematic and quantitative approaches [CL02]. Some of the most

popular performance analysis methods include queueing networks [LZGS84], Markov chains [BGDMT06], stochastic process algebras [CGHT07], and stochastic Petri nets [Mur89]. Among other common uses of such models we find, for example, capacity planning of computer infrastructures [MADD04, LFG05, CRB11], optimisations of software or infrastructure configuration [BM04, Koz10, ABG+13], what-if analysis of the system behaviour in response to software or infrastructure changes to be carried out before these are applied to the production environment [TW79, LBMAL14].

In this thesis, we focus on QoS in Big Data applications, which in recent years have become very popular and are used to support many different application domains [CZ14, IZE11, LZL+15, BGI14]. Big Data applications are typically heavily distributed across multiple commodity server nodes that collaborate to deliver one or more services. These distributed applications are usually built to be horizontally scalable, highly available, reliable and to serve requests with low latency [DMGG16]. Each application integrates these features differently characterising the software specifications. However, all of them have something in common, for example, nodes are able to communicate with each other to keep track of the state of the system, of the requests synchronisations and of the distribution of the jobs across the cluster. In addition, for some of these applications, multiple nodes are involved to process a single request. In this case, each node executes one or more tasks before the next node can continue the execution of the requests. Due to a highly interconnected infrastructure, to be able to describe these systems with mathematical tools, a performance engineer needs to solve several challenges such as reproducing the interaction of the nodes to execute one request, differentiate the different types of requests and their activity stage, define the system resource constraints, and generalise the model in a way that different system settings can be easily tested. Due to these challenges, only a limited number of performance models for Big Data applications have been developed over the last few years. In particular, our work is motivated by a significant shortage of modelling methods for NoSQL databases [AM19].

In this thesis, we focus the attention on modelling and managing QoS in NoSQL databases, and in particular we focus on a concrete implementation, namely Apache Cassandra. We choose this particular system because Cassandra has the majority of the distinguishing features of the

NoSQL approach and it is also a very popular back-end for many Big Data applications. For example, Cassandra presents an architecture which is completely decentralised with no single point of failure, it is scalable and able to process read and write requests with low latency. In addition, the consistency level required for a query can be specified directly inside the query itself, leveraging the property that all the data that this database stores is divided across the cluster nodes allowing parallelism. That is, each node stores locally part of the database and the data is replicated several times across the cluster, based on the replication factor. Therefore, different consistency levels can be used upon retrieving the data from the target nodes. Redundancy and consistency ensure tuneable data resiliency for the application so that, even under failures, the system is able to execute the requests using the available copies of the data.

**Apache Cassandra Queueing Network Model**

This thesis delivers three main research contributions. As a first contribution, we present a detailed queueing network model for Apache Cassandra that is able to predict with accuracy the database performance and to simulate all the characteristics of this software, such as different consistency levels, different replication factors and with a different number of clients into the system. After its definition, we characterise the model based on representative configuration and we analyse its performance predictions compared to the real one.

Our initial experiments are conducted on a private cloud where we demonstrate that our model is able to predict with an error below 10% some key performance metrics such as throughput, response time and CPU utilisation of the real system. We then extend our investigations with different model configurations and different hardware using a private and public cloud provider such as Microsoft Azure. We improve our performance prediction in high load, estimating in multiple points the demands and fitting them over a mathematical function to increase its accuracy.

In addition, through a case study, we demonstrate the utility of these type of models for the performance analysis of the system if a different consistency level algorithm to retrieve the data

is used by the database. Furthermore, we demonstrate that our Apache Cassandra model can also be used to predict the performance of other NoSQL databases, such as ScyllaDB, that uses a similar workflow for the execution of the queries.

**Partition-Aware Autoscaling for the Cassandra NoSQL Database**

As second contribution of this thesis, we develop a method to optimise the running costs of a NoSQL system in production. The size of the deployment of these NoSQL databases can grow very fast and it is usually related to the amount of data that the database contains and the SLA that needs to satisfy. As an example, some of the largest production deployment of Cassandra can reach thousands of nodes, such as the one used by Apple with 75000 nodes able to stores over 10 petabytes of data, or by Netflix which uses around 2500 nodes to serve over 1 trillion requests per day [Apa]. More commonly, NoSQL database deployments involve some tens of servers. The operating expenses (OPEX) to run these databases can be quite expensive especially if the resources are not utilised efficiently. As it frequently happens in production environments, the database or application does not receive a constant number of requests, but it changes over time for example as a result of the daily cycle. To optimise the number of resources involved in an infrastructure and save money, it is now common to require that a system elastically adapts the resources based on the number of active visitors.

Most of the NoSQL databases enable the possibility to the database administrator to manually add or remove nodes to the system when this is required to achieve better performance or increase the database capacity. However, in Apache Cassandra when the scaling action is invoked, several computational and network expensive processes. Indeed, to scale a cluster, the system needs to re-partition the data according to the new infrastructure and then transfers the data across the network and verifies the consistency of them. The impact of this action, in terms of computation and time, is closely related to the number of nodes involved and the amount of data that needs to be transferred across the network. For this reason, implementing an autoscaling system for a NoSQL database can be challenging. In the literature, some autoscaling methods have been published [CM13, KAB+11, DMVRT11], but they present some limitations

such as the inability to quickly adapt to new changes and/or a long time required to converge to steady-state.

As second main contribution of this thesis, we therefore present PAX, a partition-aware elastic resource management system for Apache Cassandra. PAX allows engineers to adapt NoSQL database resources to reduce operational costs without compromising the Service-Level Objectives (SLOs). Differently from existing approaches, it is not leveraging on the scaling ability of the database, but rather it dynamically changes the state of the cluster by keeping a set of nodes in sleep mode and maintaining an active set only to satisfy the performance requirements. Using a low-overhead query sampling and knowledge of the data-partitioning across the nodes, PAX automatically adapts capacity in Cassandra clusters looking for the node configuration that is able to achieve the best performance, in terms of data replicas that are available online at a given time. To ensure data consistency, PAX exploits Cassandra's hinted handoff mechanism and a shared hints storage to minimise the time necessary for a node to join the cluster. Hinted handoff is a mechanism activated by default that records data changes when one or more nodes are unavailable. We have implemented a reactive and proactive approach of PAX and compared their performance against different workloads with varying intensities and item popularity distributions, finding that the proactive version significantly reduces SLO violations. We further investigate the system optimisation using an alternative holistic approach to PAX (OPAX). Differently from the previous method, when the system needs to compare different configurations, OPAX considers the different factors together reducing, even more, the number of SLO violation.

**SD: a Divergence-based Estimation Method for Service Demands**

During our experiments, we have noticed that some of the existing demand estimation algorithms produced a set of demands that were able to predict the *mean* system performance without reproducing the same behaviour of the real system in terms of more fine-grained measures, such as state probabilities. This means that, under workloads that are sufficiently different from the reference one, the model will be likely to deliver inaccurate predictions. In addition,

existing demand estimation algorithms are designed to run only on simple test environment, which use different hardware resources, fewer nodes, and different workloads from production, which can lead to errors in the predictions when this model is applied to the production system. For these reasons, we develop a novel inference algorithm for service demand estimation in queueing network models, called State Divergence (SD) estimation.

Differently from other algorithms, SD does not consider aggregated metrics such as mean throughput or response time for its calculation, but instead uses the marginal state probability of requests at each node to infer their service demands. The marginal state probability describes the probability to observe a given system node in a specific state. Leveraging on information-theoretic divergence measures, SD infers the system service time seeking for a set of demands that reduces the distance between the marginal state probability of the real system and the one predicted by the model. In this way, the algorithm is not only able to identify the service demands to be used to predict with accuracy the system performance, but also increases the confidence that the model actually reproduces the behaviour of the real system dynamics. Moreover, since SD uses the marginal state probability to infer demands, the algorithm is also able to infer the demands of the system where jobs dynamically change classes during the request execution. This feature is essential for distributed applications to manage the different stages of the system under test and, to the best of our knowledge, only few other algorithms implement this feature but they require data that can be more difficult to obtain than the one needed in input by our SD algorithm.

The results of the experiments conducted on Apache Cassandra show that SD is able to predict with accuracy not only the performance metrics of the system under test but also to reproduce the probabilistic behaviour of the system. Furthermore, we analyse the performance of our algorithm on 100 random generated models. The results show that in the absence of noise and even just using a simplified model, the SD algorithm is able to define models that achieve even better performance reducing the error to around 1% in several cases.

## 1.2 Contributions and Thesis outline

In summary, the purpose of this thesis is to analyse and optimise the performance of distributed NoSQL databases, with a focus on the Apache Cassandra system. The main contributions are as follows:

- A stochastic queueing network model for Apache Cassandra and related NoSQL databases that is able to simulate and predict with accuracy the performance of these systems. This queueing network model is configurable to support all the main features of these systems such as the data replication across the cluster or the different consistency levels that can be applied to each query;
- A novel autoscaling method to scale efficiently NoSQL databases and able to adapt them quickly to several situations without compromising the SLO or any functionality of these systems. Through the analysis of the most frequently accessed partitions, the strategy to optimise the database performance has been analysed. Moreover, a comparison between different autoscaling strategies is presented;
- A novel inference algorithm able to estimate the demands of a system based on its state probability. The SD algorithm allows also the user to parameterize models that not only are able to predict correct performance metric but also reproduce more accurately system behaviour.

The content of the following chapters is summarised below:

**Chapter 3: Apache Cassandra Queueing Network Model**

We analyse and describe in detail the Apache Cassandra database and how these processes are translated into the presented queueing networks model. Using this model, we compare the predicted major performance of the system with different settings and configurations using private and public cloud. The results show that the model is able to predict with low error its performance. Furthermore, through a case study, we analyse the Apache Cassandra performance when a different consistency behaviour is applied to the database.

**Chapter 4: Partition-Aware Autoscaling for the Cassandra NoSQL Database**
Starting from the different approaches in the literature, we analyse them to understand the strengths and limitations of each one. Then, we present our efficient autoscaling approach able to quickly adapt the number of active nodes based on the amount and type of workload that the database is going to receive depending on the prediction and its history. In addition, we implement a workload profiling tool able to record some of the queries processed by the database with the aim of understand how frequently each partition of the database is accessed. This information is then used by the autoscaling algorithm to optimise the number of nodes available. The performance of two different autoscaling algorithms are presented. Moreover, we present the proactive and reactive autoscaling strategy and we test both on a real system. The experiments demonstrate that the proactive approach helps to not violate the SLA defined by the user. Furthermore, we investigate different autoscaling algorithm and we study a variant of the PAX system called OPAX.

**Chapter 5: SD: a Divergence-based Estimation Method for Service Demands**
Starting from the analysis of divergence of a state probability from the real one and the limitation of some existing algorithms, we develop our State Divergence algorithm. After formalising the algorithm, we conduct a sensitive analysis using Apache Cassandra to identify the best parameters to use for this minimisation problem. The parameters involved are the divergence measures, the number of state in the search space and the optimisation algorithm to use. We then analyse some metrics performance using the estimated demands, demonstrating that the algorithm is able to find demands similar to the real one that, not only generates good performance metric results, but also to maintain a comparable behaviour to the real system. We further validate our approach using 100 random models.

## 1.3 Publications

I have been made the following publication during the investigations that led to this thesis and my individual contributions of the thesis author are now described in detail for each of them:

S. Dipietro, G. Casale, and G. Serazzi. **A Queueing Network Model for Performance Prediction of Apache Cassandra**. Accepted as full paper in 10th EAI International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS'16), page 186-193, 2016.

In this paper, we present the novel Apache Cassandra queueing network model to predict the database performance. In addition, we show how the model can be used to represent other NoSQL databases and how it can be applied to analyse application changes. I contributed to this paper analysing the application and defining the presented models. Moreover, I conducted the experiments and their validation. The main results are presented in Chapter 3.

S. Dipietro, R. Buyya and G. Casale, **PAX: Partition-aware autoscaling for the Cassandra NoSQL database**. Accepted as full paper in IEEE/IFIP Network Operations and Management Symposium (NOMS 2018), Taipei, 2018.

In this paper, we present our autoscaling solution for NoSQL databases. The solution includes the workload profiler, the workload forecasting, different autoscaling algorithms and autoscaling types. I contributed to this paper defining the autoscaling's structure, understanding the fault-tolerance that Cassandra uses, implementing the autoscaling engine, strategies and the workload profiler. Moreover, I conducted all the presented experiments and wrote the tools necessary to generate the results. The main finding are presented in Chapter 4

S. Dipietro and G. Casale, **SD: a Divergence-based Estimation Method for Service Demands in Cloud Systems**. Accepted as full paper in 7th International Conference on Future Internet of Things and Cloud (FiCloud 2019), Istanbul, 2019.

In this paper, the novel demand inference algorithm reported in Chapter 5 is presented. The paper presents the SD algorithm with the sensitivity analysis and further validation using some random models. I contributed to this work defining the simplified queueing network model, the SD algorithm, running the experiment on the real system to gather the marginal probability, the sensitivity analysis and the SD validations using the real system and random models.

# Chapter 2

# Background

## 2.1 NoSQL databases

Over the last decade, human life has been completely transformed by technology in particular by computers and smart devices. As a consequence of this phenomenon, the amount of data generated by humans and machines has grown rapidly over the last 15 years and, according to Gantz [GR11], the total amount of digital data stored worldwide is going to at least double up every two years. In this new era of Big Data it has become essential to have the possibility to interrogate these data to predict or analyse historical trends before taking any decision. This new method to operate has radically changed many sectors such as finance, health, insurance, government, research, etc.

Big Data is usually defined as a huge dataset with a great variety of data types that it is difficult to process using state-of-the-art data processing approaches or traditional data processing platforms [CZ14]. The Big Data system are characterised by 5 Vs: Volume, Velocity, Variety, Veracity and Value [IA15]. These terms represents [IA15, ZE11, Lan01, Bey11, CZ14]:

- Volume: massive amount of data that the system receives or needs to handle;
- Velocity: the speed at which the data are generated and processed (for example batch, real-time, stream, etc.);

- Variety: the type of data that the system receives as input (for example structured, unstructured, probabilistic, etc.);
- Veracity: the quality of the data received. Better data as input could produce better analysis or results;
- Value: is the most important aspect and represents how the business can take advantage of them like, for example, make profit from this data and repay the investments.

This Big Data era opens several opportunities produced by the data-intensive decision-making such as define better strategy directions, identify new products and services, enhance the customer experience, identify new possible markets, etc. [MH13, CZ14, MCB+11, AHLJ12]. On the other hand, it creates more computing challenges to be solved especially in the receiving, processing and storing processes. A usual path for this data involves several stages like, data pre-processing to format the input data that are inconsistent or incomplete. So the data goes through a data cleaning process, data integration, data transformation and data reduction [CZ14, HC06]. After the pre-processing phase, the data is ready to be the analysed. Depending on the type of application, the data can be processed in real-time (or near-time) [Man04] or stored to be analysed later with batch processing tools [DG04, ZCDD12]. The techniques applied can be different including statistic, data mining, machine learning, neural networks, optimisation methods [CZ14].

Independently from the kind of applications, the data and the results of their execution need to be stored. These applications are able to collect and analyse a huge amount of information on the order of petabytes, exabytes or zettabytes of data and they need to retrieve them with low latency. The usage of Relational DataBase Management System (RDBMS), such as MySQL[MyS], SQLServer [SQL], Oracle [Ora], etc., is not suggested for these kind of applications because, due to their construction, they are not able to scale and retrieve efficiently such amount of data. In addition, they are not able to store unstructured data. For this reason, NoSQL databases have been developed.

"Not Only SQL" (NoSQL) are distributed databases that are able to manage a large amount

of data. As also the name suggests, they support Standard Query Language (SQL) but they are not entirely relational and, for this reason, some operations like *join* cannot be performed on NoSQL databases. NoSQL databases avoid the rigidity of relational databases separating, in two independent parts, the data storage and management. In this way, NoSQL databases are able to store also unstructured and not relational data [HHLD11]. This design allows the data storage to focus on the scalability and high performance of the database and the data management to be optimised for the low-level operations [CDG+06, CZ14].

Over the years, different types of NoSQL databases have been developed, each one with specific characteristic based on the data model, architecture and properties. Among the most popular, we can find BigTable[CDG+06], HBase[HBa], Hypertable[Hyp], MongoDB[Mon], BerkeleyDB [OBS99], Redis[Red], Voldemort[SKG+12], CouchDB[Cou], SimpleDB[Sim], Riak[Ria] and Cassandra[LM10].

Usually, RDBMS databases ensure to the user the consistency of the database basing its transactions execution on Atomicity, Consistency, Isolation and Durability (ACID) [Gra81]. However, since the ACID characteristics are not applicable to NoSQL databases, this theorem has been generalised defining the CAP theorem [GL02]. The CAP theorem is based on Consistency, Availability and Partition Tolerance. It postulates that a system can only have two of these three factors [Bre00, GL02]. These factors are defined as [BL13]:

- **Consistency**: all the cluster nodes can see and retrieve the same version of the data at the same time;
- **Availability**: the system guarantees that the client receives an answer regardless of the state of the system. In other words, the system is able to cope with hardware or power failure and network problems;
- **Partition-tolerance**: the system distributes the data and/or the requests across the cluster to improve the overall performance. Sharding is a popular technique to distribute slice of data on separate nodes to increase the throughput.

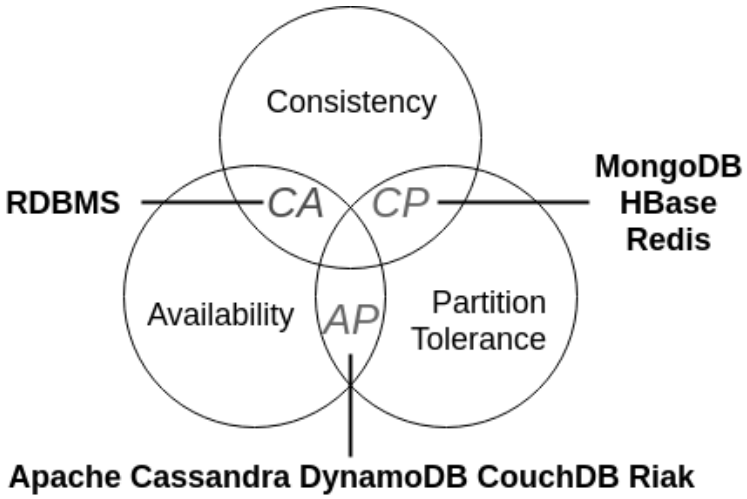



Figure 2.1: CAP Theorem representation with some databases examples.

Since only two over these three factors of the CAP theorem can be chosen at the same time, different NoSQL databases were developed with a different balance of these factors based on the importance that each product gives them [HHLD11]. As shown in Figure 2.1, among the most well known databases, we have databases like BigTable [CDG+06], HBase [HBa], Hypertable [Hyp], MongoDB [Mon], BerkeleyDB [OBS99], Redis [Red], that focus their attention on the Consistency and the Partition tolerance and databases like Voldemort [SKG+12], CouchDB[Cou], SimpleDB [Sim], Riak [Ria] and Cassandra [LM10] that focus their attention on the availability and the partition tolerance. On the other hand, RDBMS databases are based on consistency and availability without considering the partition tolerance since they frequently store all the data locally.

Among each group of databases, another important classification can be done based on the type of data that they are able to store. Four different types of input data can be used by a NoSQL database:

- **Key-value**: each value stored in the database is associated with a key. This key is usually hashed to have a higher distribution of the keys around the different nodes of the database. This kind of databases is more suitable for retrieving small values as user profiles, sessions, products name or shopping carts. Some examples are Amazon Dynamo, Voldemort and BerkleyDB;

- **Column-oriented**: it is designed to store multiple attributes per key. It is used especially when it is necessary to keep versioned data or batch-oriented data processing. Some examples are: BigTable, HyperTable, Cassandra, SimpleDB;

- **Document-based**: it is designed to contain semi-structured data such as XML, JSON or BSON. Respect to the key-value, the fields are fully searchable (key and value). Some examples are: MongoDB, CouchDB.

- **Graph database**: It is designed to keep the interconnection between the key-value pairings in the database. In general, this type of databases is useful when the interest is more on the relationship between the objects than the data itself [MH13]. An example is Neo4j[neo].

The choice of the correct NoSQL database to use for a specific application is based on these two factors. For the purpose of this thesis, we focus our attention on Apache Cassandra NoSQL database. It is a column-oriented database that is characterised by a good availability and partitions of the data in preference to consistency.

## 2.2 Cassandra

Cassandra is one of the most popular NoSQL databases. Initially developed by Facebook to support one of their products (Facebook Inbox Search), it was soon deployed also to other Facebook's products. Today, it is an open source software under the Apache licence, supported by Datastax [Dat], and used for thousands of applications.

The initial idea of Lakshman and Malik when they started to develop Cassandra was to create a distributed storage system for managing a huge amount of structured data, distributed across many commodity servers without a single point of failure [LM10]. The architecture of Cassandra may be seen as a combination of other two well-known NoSQL databases: Google BigTable and Amazon Dynamo. Indeed, like in Amazon Dynamo, all nodes are disposed in circle without any concept of master-slave and the overall key space is divided into equal parts among the nodes,

using a consistent hashing algorithm [MRSJ14]. Differently, Dynamo is a key-value store while Cassandra uses a column-oriented approach like BigTable. The columns are grouped together into sets, called "column families".

The main architectural and data modelling feature of Cassandra are:

- **Schema-less**: even if Cassandra is a column oriented database, the columns are very flexible and it is not mandatory to define a specific structure for the rows. This implies that different numbers of columns can be stored in different rows and new columns can be added anytime into the table.

- **Data replication**: the data are stored locally in each node and the data replication is achieved storing multiple time the same partition across different nodes. This is different from other NoSQL database where replication is provided by the file system.

- **Performance**: Cassandra is optimised for write operations. This means that it is able to reach low latency for write operations. Cassandra is also able to reply with low latency to read requests using some caching mechanism and optimising the access to the storage disks.

- **Fault tolerant**: it is built to be deployed on commodity hardware or in the cloud infrastructure. This implies that the machines can be turned off or fail at any time. Since the data is automatically replicated to multiple nodes or across multiple data centers, if few nodes of the cluster failed at the same time, they can be manually replaced with no downtime.

- **Decentralised**: There are no single points of failure and every node in the cluster is identical. In addition, all the nodes accept queries from the client, so there are no network bottlenecks.

- **Scalable**: the database performance scales linearly with the number of nodes in the cluster. If more storage or performance (read and write throughput) are required, new nodes can be always added to the cluster.

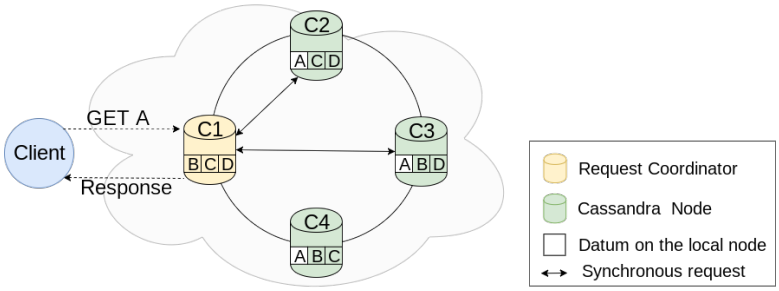



Figure 2.2: Cassandra read request representation.

- **Durable**: Cassandra is suitable for applications that cannot afford to lose data, even when an entire data-center goes down.

In the next sections, we describe how all these features have been integrated in Cassandra and the required architecture to support them.

### 2.2.1 Cassandra architecture

A normal Apache Cassandra deployment is represented in Figure 2.2. The figure represents a cluster of four nodes disposed in circle since Cassandra does not differentiate the nodes in master and slaves. To deploy an infrastructure with no single point of failure, each Cassandra node is able to perform all the processes so, there is no separation of duties across the cluster. Each node is able to: connect itself to the rest of the cluster, retrieve information about the state of the system using the *gossip protocol*, allow clients to connect to it to perform queries, retrieve data across the cluster, provide data synchronisation and store part of that database on its local disks.

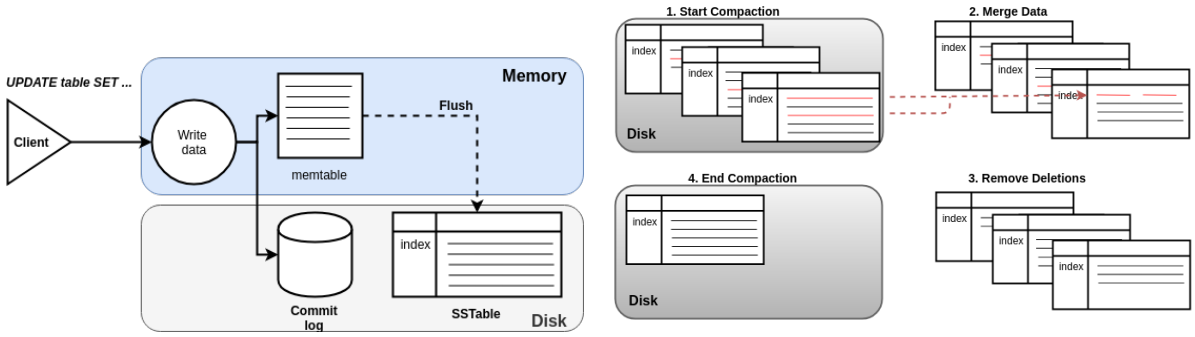
When the cluster starts for the first time, inside the configuration file of each node, a set of IP addresses of some existing nodes of the cluster are specified. These nodes are called "*seeds*" and they are used from the other nodes to retrieve initial information, IP addresses of the other machines and join the cluster. During the join process, part of the hashing space is attributed to each node. The overall hashing table (from $-2^{63}$ to $+2^{63}-1$) is split in smaller ranges based on the overall number of partitions, called virtual nodes (*vnodes*). A vnode is

defined as a hash range on which all the data that has the hashed primary key in that range are stored. These ranges can be defined by the user inside the Cassandra configuration file of each machine, or automatically attributed by the program. By default, Cassandra creates 256 vnodes for each node to ensure an even distribution of the data and its copy across the nodes. The default Cassandra partitioner is Murmur3Partitioner [App19] which use the MurmurHash hashing algorithm to hash the primary keys. This algorithm has been chosen because it provides a fast non-cryptography hashing function and more equal distribution of the data across the vnodes. In Figure 2.2, the partitions' distribution has been simplified and each partition is represented with capital letters inside each node.

To ensure high availability, durability of the data and reduce the database latency, each vnode can be stored multiple times inside the same cluster. The number of copies is defined by the *Replication Factor* (*RF*) and it is specific for each keyspace of the database. For example in Figure 2.2, the vnode has been represented with capital letters and the replication factor applied to this keyspace is 3, since the cluster stores three copies of the same data partition.

Another main duty of each Cassandra node is to resolve queries from the clients. The client is able to connect to any of the Cassandra nodes of the cluster and once the connection is established, it can retrieve the complete list of the nodes available and it can choose to which node forward its query. The selected node for the query becomes responsible for it and it is called *request coordinator*. The request coordinator has the duty to perform all the processes necessary to complete the query.

One of the Cassandra peculiarity is that for each query it is possible to define a different *Consistency Level*. The Consistency Level defines the number of copies that need to be read or written to complete a query. Three are the main Consistency Levels: ONE, QUORUM, ALL. With Consistency Level ONE, only one copy of the data needs to be read or updated to complete the query. So the request coordinator chooses one node that contains that data to perform the operation. When the response is received back from the request coordinator, the request can terminate returning the result to the client. This is the lowest consistency level and also the one which ensures less consistency of the data. On the other hand, the Consistency




(a) Write operations (b) Compaction

Figure 2.3: a. Representation of write operations in Cassandra; b. Cassandra compaction processes.

Level ALL involves all the copies of the data, so the number defined as $RF$, and provide the best consistency ensuring that the read data is the latest one or that the data is written on all nodes. An intermediate possible Consistency level is the QUORUM which involves $(RF/2+1)$ copies.

Choosing the right Consistency Level and Replication Factor is always a balance between performance and data consistency. To achieve *strong* consistency, the user can either use the Consistency Level ALL or have an overlap between the read and write consistency levels. This can be defined with $W + R > RF$ where $W$ and $R$ are the respectively write and read consistency level and $RF$ the Replication Factor used. Let's assume that we use for read and write operations the Consistency Level QUORUM, this guarantees that the latest version of the data is always read achieving a strong consistency. On the other hand, if strong consistency is not required by the application, lower consistency level can be used achieving also better performance [WLZZ14, LRS+14].

### 2.2.2 Read and Write operations

Cassandra is a low latency database optimised for write operations but able to achieve good performance also for read operations [LM10]. Similarly to BigTable, to increase the overall throughput, Cassandra uses an in-memory data structure, called Memtable. As shown in Figure 2.3, after a write operation reaches the database Cassandra, firstly, writes the query

into the request coordinator commit log (on disk) for durability and then it writes the data on the Memtable and informs all the other nodes that hold a copy of the data of the update. When the consistency level applied to the query is satisfied, the request coordinator can notify the client that the operation is successfully executed.

When the Memtable size reaches a certain threshold, defined in the configuration file, Cassandra flushes the table on the node local disk in form of SSTables (Sorted String Table). This process is called "*minor compaction*". The SSTables are *immutable* and written *sequentially* into the disk. Besides the data, each SSTable contains also some metadata information as a key index and a bloom filter to rapidly and memory-efficiently check whether a key is included in a specific SSTable file. The minor compaction has also the capability to remove the previous version of the data, called tombstones, from the new SSTable. Since, over time, the data grows or has been updated, previous values can decrease the database performance. To enhance the database performance and to reduce the searching time, Cassandra performs a merge of all the SSTables. This process is called "major compaction" and it needs to be run by the system administrator manually since it is an expensive operation to perform.

Differently, when a read operation needs to be performed, the request coordinator contacts as many nodes that hold that primary key as necessary to satisfy the Consistency Level of the requests. The decision on which nodes to contact is based on the average latency of the previous requests. When all the contacted nodes return the value, the request coordinator compares the results and responds to the client. If some incoherence is detected, the request coordinator informs the not updated nodes of the latest value for that record, similarly to a write operation. Each node retrieves the data looking, initially, into the Memtable to see if it contains the value and then, if necessary, it reads the SSTables to gather the value from disk. To speed up the data search on SSTables, Cassandra maintains in memory all the information regarding the SSTables and the column indices, which allows to jump directly to the right chunk of the disk [LM10].

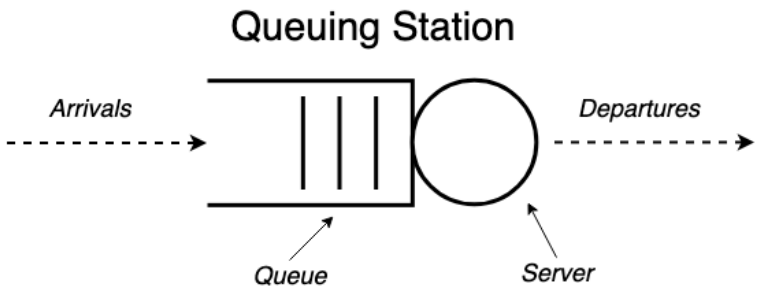



Figure 2.4: Queueing Station

## 2.3 Queueing Networks

Queueing theory is a mathematical study of the delays and waiting time. The system that needs to be studied is composed of a series of stations that interact together and compose the queueing network. In computer science, these stations usually represent the different resources of the system like, for example, CPU, memory, disk or an entire server. The jobs are the entities which visit the different stations and simulate how a request is executed by the system.

A station is usually represented as in Figure 2.4. When a job reaches a station, it is inserted into the *queue* of the station and it waits there until the *server* chooses it to be executed. After the job is executed with a user defined service time, the job leaves the station to reach another component of the model. Different policies and servicing strategies can be applied to each station.

In order to describe the characteristic of the represented resource, we need to know the details of:

- *Arrival rate* $\lambda$: the mean value of jobs that arrive at the resource. Not only the mean value is important but also the distribution function of the inter-arrival times. Regarding the inter-arrival distribution, we mainly considered the exponential distribution.
- *Service Time* $\theta$: the mean time of a job to be executed by the server. As for the arrival rate, the service time is characterised by a mean value and a distribution. Also in this case, if not specified differently, the main service time distribution used is exponential.
- *Number of servers* $c$: This indicates how many jobs can be processed in parallel by the

station at the same time. Three type of cases can be considered: *single-server*, *multi-server* and *infinite-server*. In the case of single-server, the server has the capacity to process a single job at the time. Differently, in case of multi-server with $c$ cores, the same amount of jobs can be processed at the same time. In these first two cases, the jobs that are not currently processed wait in the queue to be executed. The method to decide which job should be executed next, is defined by the *service discipline*. Infinite-server is a special case of the multi-server in which the number of servers is infinite. In this case, the job is not spending time in the queue since a server is always available to process the job.

- *Queue Capacity*: if all the servers are processing some other requests when a new request arrives at the station, the new request waits in the queue. If the arrival rate of the job is bigger than the service rate, there is the possibility that the queue becomes full. In this case, the queue has two possibilities: notify to the arrival process that the queue is full and the arrival process is suspended until the queue has same spare capacity or drop the jobs that cannot be taken by the queue. However, usually, the queue is considered infinite and so it is not necessary to apply any policy.

- *Population*: number of jobs that are circulating on the model. If the number of jobs $N$ is fixed, no more than $N$ requests can be seen by the resource. The jobs that share the same behaviour can be grouped together in *classes*. Different classes usually have different characteristics such as behaviour or service demand.

- *Service Discipline*: is the method that decides which is the next job to be executed in case that more than one job is waiting in the queue. The service disciplines can be:

  - First Come First Served (FCFS): based on the arrival order (FIFO);
  - Last Come First Served (LCFS): the last job arrived is the first to be executed (LIFO)
  - Random: the job is selected randomly from the queue
  - Priority: the decision is based on the priority assigned to the class of the jobs in the queue.

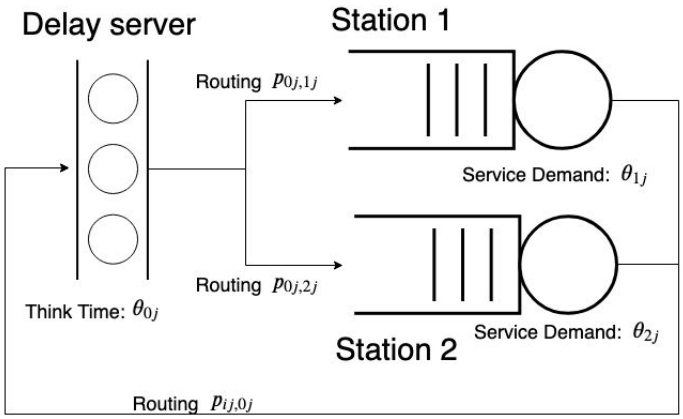



Figure 2.5: Closed queueing network

For the purpose of this thesis, we focus our attention to a particular subset of networks, called closed queueing networks. Differently, for the open queueing network, the number of jobs $N$ inside the model is always constant.

### 2.3.1 Closed queueing network

To understand better how a closed queueing model is structured, in Figure 2.5 a simple model is presented. The model describes a system with two resources, represented by the queueing stations one and two. These two resources illustrate the behaviour of, for example, two different servers that process some web requests in parallel. The requests are generated by a particular type of queueing station, called *infinite server* (or delay server). Usually, it simulates the user behaviour and it is the starting and ending point of the requests for a closed queueing network.

The requests generated by the infinite server are then distributed between the two queueing stations based on some routing probability $p$. The requests that reach one of these two queues are, first, inserted into the queue of the station waiting for their turn to be processed and then executed. The requests in the server can be executed in two modes: *preemptive* and *non-preemptive*. In non-preemptive mode, the server needs to execute the job entirely before being able to serve the next one. Differently, with the preemptive mode, the server scheduler is able to suspend the current job to execute another one. Normally, the preemptive mode is used to simulate the real CPU behaviour. To be categorised as closed model, the requests need to be

able to reach the starting point after they have been processed by the last queueing station.

Depending on the level of detail and the complexity of the system under test, the queueing network models can be divided in *single-class* or *multi-class*. Single-class models are usually used to represent simple or a higher level view of the systems. Moreover, they are less expensive and quicker to be analysed but the results can be inaccurate. Nowadays, the majority of the queueing networks are multi-class due to the complexity of the analysed systems and the higher accuracy that is possible to gather from them. All the queueing models presented in this thesis are multi-class models.

After designing the model, before being able to solve it, it is necessary to parametrize it. During the parametrisation phase of the model, for each class at each station, a demand value and distribution needs to be defined. In the case of the infinite server, the demand is called *thinking time*. Apart from the demands, some other value needs to be defined. Let's consider a closed queueing network model with $R$ classes and $K$ queueing stations both in the range of $1 \leq r \leq R$ and $1 \leq i \leq K$ . So, to be able to analyse or simulate this model, it is necessary to define:

- $N_r$: number of jobs of class $r$ that circulates in the system. The total number of jobs is fixed and equal to $N = \sum_{r=1}^{R} N_r$ but $N_r$ can variate between classes;
- $p_{ir,i'r'}$: probability that a job of class $r$ at station $i$ reaches the station $i'$ with the class $r'$;
- $v_{ir}$: average number of visits of class-$r$ jobs at station $i$;
- $\theta_{ir}$: service demand of class $r$ at station $i$;
- $Z_r$: the think time for jobs of class $r$. This is usually an average between the time past from the completion to the generation of the same job of class $r$;
- $\mu_{ir}$: service rate of class-$r$ jobs at station $i$. It is the inverse of the service time: $\mu_{ir} = \theta_{ir}^{-1}$;

Once the model is defined and parametrised, in order to retrieve some performance metrics, it is necessary to solve it. In the next sections, we present first some solutions and performance

| Notation | Description |
|---|---|
| $M$ | Number of stations in the model (queueing and infinite server) |
| $K$ | Number of queueing stations. |
| $C$ | Number of job into the infinite servers ($J = M - K$). |
| $R$ | Number of classes. |
| $N_r$ | Number of jobs in class $r$, $N = \sum_r N_r$ |
| $\theta_{ir}$ | Service demand of class $r$ at node $i$. |
| $Z_r$ | Think time for job of class $r$ |
| $\sigma_r$ | Response time of class-$r$, $\sigma_r = \sum_{k=K+1}^{M} \Theta_{kr}$ |
| $c_i$ | Number of servers at node $i$ |
| $s_{ir}$ | Service time at node $i$ for jobs in class $r$ |
| $v_{ir}$ | Visit ratio at station $i$ for jobs in class $r$ |
| $p_{ir,i'r'}$ | Routing probability for job in node $i$ of class $r$ to node $i'$ in class $r'$ |
| $\pi(\boldsymbol{n})$ | Probability to observe the system in state $\boldsymbol{n}$ |
| $\pi_i(\boldsymbol{n}_i)$ | Probability to observe the station $i$ in state $\boldsymbol{n}_i$ |

Table 2.1: Summary of main notation for the model input parameters

metrics for closed network and then the most popular methods to analyse closed network models.

### 2.3.2 Analysis and Solutions

Over the years, several methods to solve these models have been presented. These include several analytical and stochastic analysis methods and simulation. In general, the analysis of a queueing network model is amenable to the analysis of a Continuous-Time Markov Chain (CTMC). CTMC analyses the state of the system at each moment where the state is represented by the distribution of the jobs at each station of the different classes.

Given a model with $R$ classes and $M$ stations which $K < M$ are queueing stations, the state of the $i$-th station can be defined as $\boldsymbol{n}_i = (n_{i1}, n_{i2}, ..., n_{iR})$. Moreover, the state of the entire system can be defined as $\boldsymbol{n} = (\boldsymbol{n_0}, \boldsymbol{n_1}, ..., \boldsymbol{n_K})$ and we require $\sum_{\boldsymbol{n} \in \boldsymbol{S}} \pi(\boldsymbol{n}) = 1$, where $\boldsymbol{S} = \{\boldsymbol{n} \in \mathbb{N}^{MR} | n_{kr} \geq 0, \sum_{k=1}^{M} n_{kr} = N_r\}$ is the model state space. We also define the job population as $\boldsymbol{N} = (N_1, ..., N_R)$ and the demands vector for all classes and stations as $\boldsymbol{\theta} = (\theta_{11}, ..., \theta_{MR})$. Observing this model, we can calculate some system properties at equilibrium (or steady-state) such as [BCMP75, Har04, GN67]:

- *Joint state probability* $\pi(\boldsymbol{n})$: describes the probability to observe the system in the state $\boldsymbol{n}$
- *Marginal State probability* $\pi_i(\boldsymbol{n}_i)$: describes the probability that the station $i$ is the state $\boldsymbol{n}_i$.

The calculation of the joint and marginal state probability can be achieved using the product-form solution. For the purpose of this thesis, we focus in particular on product-form queueing network models since their steady-state probabilities can be analytically calculated [BCMP75, Har04, GN67]. Under specific assumptions given by the BCMP theorem [BCMP75], these models admit, up to a normalising constant, a closed-form solution for their equilibrium distribution from which marginal probability expressions can be explicitly derived. The BCMP theorem [BCMP75] considers queueing network models composed by nodes that satisfy the following assumptions:

- Nodes admit either a First Come First Served (FCFS), Processor Sharing (PS), Infinite Server (IS or delay) or Last Come First Served with PRemption (LCFS-PR)
- Service time distributions at nodes are exponentially distributed at FCFS stations
- Load dependent service rates are supported, but under FCFS scheduling the demand can depend only on the total number of jobs at the node.
- Arrival processes: where, for open networks, the arrival process is Poisson and all jobs arrive at the network from a single source or where different sources stream jobs to different chains.
- Routing between nodes is governed by a discrete-time Markov chain that specifies the probability that a job departing node $i$ in class $r$ joins node $j$ in class $s$.

For the models that complains to the BCMP theorem, the joint state probabilities with the product-form can be calculated like [BCMP75]

$$\pi(\boldsymbol{n}) = \frac{1}{G_{\boldsymbol{\Theta}}(\boldsymbol{N})} \prod_{i=1}^{M} f_i(\boldsymbol{n}_i) \qquad \boldsymbol{n} \in \boldsymbol{S} \tag{2.1}$$

where $G_{\boldsymbol{\Theta}}$ is a normalising constant and the analytical form of $f_i(\boldsymbol{n}_i)$ depends on the type of scheduling policy assumed at station $i$.

Let's consider for example a closed model where the scheduling policy for the target station is preemptive (or processor sharing (PS)). If the $K$ stations are single-server, the product-form formula for the steady-state probability of this model is given by

$$\pi(\boldsymbol{n}) = \frac{1}{G_{\boldsymbol{\Theta}}(\boldsymbol{N})} \prod_{i=1}^{M} n_i! \prod_{k=1}^{K} \prod_{r=1}^{R} \frac{\theta_{kr}^{n_{kr}}}{n_{kr}!} \qquad \boldsymbol{n} \in \boldsymbol{S} \tag{2.2}$$

The formula is different if we assume that PS node $i$ is multi-server and it has $c_i \geq 1$ servers, such that when the node has up to $c_i$ running jobs they run without contention. Note that infinite server nodes may be seen as a special case of multi-server nodes where $c_i = +\infty$. In this case, the product-form formula for the steady-state probability of this model is given by

$$\pi(\boldsymbol{n}) = \frac{1}{G_{\boldsymbol{\Theta}}(\boldsymbol{N})} \prod_{i=1}^{M} n_i! \prod_{k=1}^{K} \prod_{r=1}^{R} \frac{\theta_{kr}^{n_{kr}}}{n_{kr}! \cdot \prod_{u=1}^{n_k} \min(u, c_i)} \qquad \boldsymbol{n} \in \boldsymbol{S} \tag{2.3}$$

where the factorials capture the ordering of jobs within the PS node and the product of $\min(u, c_i)$ functions describes load-dependence due to the multi-server nodes. The infinite server is a special case of these of Equation (2.3) that it is achieved removing the limitation of number of servers. So, $c_i$ is set to be $N$.

From Equation (2.2) and (2.3), the normalising constant $G_{\boldsymbol{\Theta}}(\boldsymbol{N})$ may be determined by requiring that state probabilities sum to one, which leads to the explicit formula

$$G_{\boldsymbol{\Theta}}(\boldsymbol{N}) = \sum_{\boldsymbol{n} \in \boldsymbol{S}} \prod_{i=1}^{M} n_i! \prod_{k=1}^{K} \prod_{r=1}^{R} \frac{\theta_{kr}^{n_{kr}}}{n_{kr}! \cdot \prod_{u=1}^{n_k} \min(u, c_i)} \tag{2.4}$$

However, computing $G_{\boldsymbol{\Theta}}(\boldsymbol{N})$ using direct summation over the state space $\boldsymbol{S}$ is usually infeasible unless the model is small, because the number of states generated by the model grows as $\mathcal{O}(N^{MR})$, where $N$ is the total number of jobs in the model. To solve this problem, several approaches have been proposed using exact [BM93, RK75, CG86, Gou56, HL04, RL80, Cas11] or approximate methods [Sch79, Sau02, CHW75, PPSC13] when $c_i = 1$. However, few of these

techniques are applicable to the general multi-server case $c_i > 1$ [SSV07, CKOG10, NKY+11], therefore posing a challenge in describing PS nodes with multiple servers, which are yet often needed to describe contention at multi-core processors.

Using these product-form equations, it is also possible to deduce model performance metrics such as throughput, response time, utilisation, and mean number of jobs [BGDMT06]. For example, the class-$r$ throughput is given by

$$X_r = \frac{G_{\boldsymbol{\Theta}}(\boldsymbol{N} - 1_r)}{G_{\boldsymbol{\Theta}}(\boldsymbol{N})} \tag{2.5}$$

for $r = 1, \ldots, R$, where $\boldsymbol{N} - 1_r$ represents a vector obtained from $\boldsymbol{N}$ by decreasing of one job the population of class $r$. The same performance metrics can also be calculated using the marginal state probability such as

- Utilisation $U_{ir}$: the average server utilisation of the class-r job at the station $i$

$$U_{ir} = \sum_{\substack{\boldsymbol{n} \in s \\ \& \ n_{ir} > 0}} \pi_i(\boldsymbol{n}_i)$$

- Station Utilisation $U_i$: the overall utilisation of the $i$th station

$$U_i = \sum_{r=1}^{R} U_{ir}$$

- Throughput $X_{ir}$: the average rate of completions of class-$r$ jobs at station $i$

$$X_{ir} = U_{ir}\mu_{ir}$$

- System Throughput $X_r$: the average system throughput at the reference station for the job of the class $r$

$$X_r = \frac{X_{ir}}{v_{ir}}$$

- Mean Queue length $Q_{ir}$: the average number of class-$r$ jobs at the station $i$. This include

also the jobs that are in execution on the station

$$Q_{ir} = \sum_{\substack{\boldsymbol{n}\in S \\ \& \ n_{ir}>0}} n_{ir}\pi_i(\boldsymbol{n}_i)$$

- Mean Response time $W_{ir}$: the average time a job of class $r$ waits to the station $i$ before being executed

$$W_{ir} = \frac{Q_{ir}}{X_{ir}}$$

- System Response time $W_i$: the average time spent a job of class $r$ to return to the reference station and be processed again. This is calculated summing all the mean response times for the job of class-r and weight them with the visit at each station

$$W_r = \sum_{i=i}^{K} W_{ir} v_{ir}$$

### 2.3.3 Class-switching models

The BCMP theorem allows within the described class of models also to enable class-switching, whereby jobs departing from a node in class $r$ can arrive in the destination node in class $s \neq r$. However, under class-switching, the number of jobs in each class is no longer constant over time, implying that the underpinning state space is much larger than in regular multiclass networks without class switching. To address this issue, under class-switching the original $R$ job classes can be partitioned into $C \leq R$ disjoint *chains* [Zah79, BBS$^+$77, Bru78, Won78] , defined by the strongly connected components of the class-switching probability matrix. In other words, chains are defined such that a job within chain $c$ can become over time any of the classes in that chain, but cannot become of a class belonging to any other chain $h \neq c$.

To do so, we first need to calculate the number of visits ($v_{ir}$) to each station and class of the original network and then compute the corresponding number of visits for each chain $q$ at every

node $i$ in the aggregate model ($\hat{v}_{iq}$) as

$$\hat{v}_{iq} = \frac{\sum_{c \in C_q} v_{ic}}{\sum_{c \in C_q} v_{1c}} \tag{2.6}$$

where we assume that station $i = 1$ is the reference station for the computation of the system throughput for each class.

Thanks to this transformation, the class-switching model can now be solved as a standard multiclass model with job populations equal to the chain population. However, due to this transformation, the information regarding the service time of the different classes at a station is lost by the aggregation. Let $q$ now represent the class in the new model associated to the original chain. The service demand for class $q$ at station $i$ needs to be computed as [BGDMT06]

$$\hat{\theta}_{iq} = \sum_{r \in C_q} \alpha_{ir} \theta_{ir} \tag{2.7}$$

where $\alpha_i$ is a scale factor defined as

$$\alpha_{ir} = \frac{v_{ir}}{\sum_{c \in C_q} v_{ic}} \tag{2.8}$$

The aggregate demands $\hat{\theta}_{iq}$ are stored in matrix $\hat{\mathbf{\Theta}}$. The population of chain $q$ is set to $\hat{N}_q = \sum_{c \in C_q} N_c$.

## 2.4 Demand estimation algorithms

Demand estimation is one of the most challenging steps for performance model parametrisation. Existing techniques are based mostly on the statistical inference of indirect measurements such as throughput, response time and resource utilisation. The demand estimation algorithms are grouped and presented based on the technique used: regression, machine learning, maximum likelihood estimation and optimisation.

### 2.4.1 Regression

Linear regression is a popular statistical method for the service demand inference. Given a set of independent variables, in regression called *control variables*, $x_1, .., x_r$ and a dependent variable, called *response variable*, $y$, the linear relation between them can be defined as:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + .. + \beta_r x_r + \epsilon \tag{2.9}$$

The goal of this method is to identify a set of parameters $\beta_j$ with $0 \leq j \leq r$ that minimise the *residual error* $\epsilon$. Using the Utilisation Law provided by the Queueing Network theory, the linear regression problem can be rewrited as:

$$U^{(n)} = U_0^{(n)} + \sum_{r \in R} X_r^{(n)} \theta_r + \epsilon^{(n)} \qquad n = 1, .., N \tag{2.10}$$

where $U$ is the resource utilisation, $X_r$ the throughput of class $r$ with $0 \leq r \leq R$ and $\theta_r$ represents the demand for each class of the system. The index $n \in N$ refers to the observations considered. To solve the set of equations with $R$ control variables, a system of $N > R + 1$ observations are necessary to be collected from the real system [CCT08]. The error of the current set of $\theta$ values can be evaluated using, for example, the Least Squares (LSQ) regression or the non-negative Least Squares (NNLS) regression. The estimate demand $\theta_j$ is the mean values across all the $\theta_j^{(n)}$ observations.

The regression method applied to service demand estimation has been presented, for the first time, by Bard and Shatzoff in 1978 to characterise the resource consumption of some specific functions of an operating system [BS78]. Then, the method has been extended in [RV95] to estimate the CPU demand for a single-thread application using multiple classes. [PSST08] and [ZCS07] apply the regression methodology to estimate the different demands of more complex systems such as a multi-tier architecture. In particular, [ZCS07] conducts an extensive evaluation using the TPC-W e-commerce platform with different workloads and demonstrates the robustness of this algorithms.

However, this method can fail or produce wrong results if the observation time is too small, with outliers or if the variance across the samples is limited [RV95, ZCS07]. In addition, a well-known issue which affects the regression method is the multicollinearity [MPV12, KKRR11]. Multicollinearity occurs when two or more explanatory variables in a multiple regression model are highly linearly related. This can produce unstable and unreliable results. To address these problems, more robust statistical methods, such as Least Absolute Differences (LAD) [KZ06, ZCS07, SKZ07] and Least trimmed squares (LTS)[CCT08] can be adopted. Other techniques involve the use of machine learning algorithm, such as Support Vector Regression (SVR) [KKRR11] to achieve the same scope.

### 2.4.2 Machine Learning

Machine learning algorithms have also been exploited in demand estimation. One of the first and most used technique is based on the Kalman filter [SCBK15, ZM]. The Kalman filter technique estimates the *state* vector $\boldsymbol{x}$ from a series of measurements $\boldsymbol{z}$. The term state is defined as the complete representation of the system status at a given time of a dynamic system that evolves over time. The Kalman filter is defined by two equations. The first equation describes how the system has evolved from the previous state $(k-1)$. So, the current state $\boldsymbol{x}_k$ is calculated as:

$$\boldsymbol{x}_k = \boldsymbol{F}_k \boldsymbol{x}_{k-1} + \boldsymbol{G}_k \boldsymbol{u}_k + \boldsymbol{w}_k \tag{2.11}$$

where the $\boldsymbol{F}$ and $\boldsymbol{G}$ matrix are the state transition and control-input model, $\boldsymbol{u}$ contains the input from the system and $\boldsymbol{w}$ the process noise. The second equation, defines the linear relation between the current system state $\boldsymbol{x}_k$ and the measurements $\boldsymbol{z}_k$.

$$\boldsymbol{z}_k = \boldsymbol{H}_k \boldsymbol{x}_k + \boldsymbol{v}_k \tag{2.12}$$

where $\boldsymbol{v}_k$ is the observation noise and $\boldsymbol{H}_k$ the observation matrix which maps the state to the observation space.

To use this method also for non-linear problems, the Extended Kalman Filter (EKF) has been

developed and its usage has been demonstrated in [Sim06]. The EKF has been applied in [ZYW+05, ZWL08, WZL05, WZL06] to estimate the demands of a real-time system of an open and close model with a single class. [KZT] and [WHQ+12] extended the Kalman filter to multi-class model.

Another machine learning technique for demand inference includes clustering [CS14], which observes the data composed by timestamps, throughput and utilisation. [CDS10] develops furthermore this technique recognising the deviation over time of the demands, such as those resulting from hardware upgrades. A similar approach has been also applied in [KYTA12] but using pattern recognition techniques. Differently, [SBC+08] applied the independent component analysis to profile the workload using CPU utilisation and the number of customers information.

### 2.4.3 Maximum Likelihood Estimation

MMaximum Likelihood Estimation (MLE) is a statistical method based on the likelihood function $\mathcal{L}(\theta \mid x)$ that, starting from some observations $x$, infers the parameters $\theta$ that categorise the normal distribution. The output of this function is a set of particular parameters $\theta$ that obtained the highest likelihood value. The MLE can be described as:

$$\hat{\theta} = max\ \mathcal{L}(\theta \mid x) \tag{2.13}$$

Recently, [PCPS15, PPSC13, KPSCD09] propose an algorithm that considers as input the response time of the requests. However, this information can sometime be difficult to retrieve or to save in a log without impacting the application performance. In addition, [KKRD12] introduces the concept of confidence in the demand estimation with this method. Differently, [WCKN16] propose the QMLE algorithm that uses mean queue length values, rather than response times, to perform demand estimation. The queue-length is also used by other algorithms such as Gibbs sampling and Bayesian inference [SJ11, WC13]. However, such methods are computationally intensive and they can require even hours to complete compare to other algorithm that are able to provide a result in minutes.

### 2.4.4 Optimisation

The optimisation problem is a method that looks for a set of parameters able to minimise (or maximise) an objective function $f$ within a given domain $D \subseteq \mathbb{R}^n$ and a set of constraints $\Omega \subseteq D$. Then, an optimisation problem that minimise a function can be written as:

$$\min_{\Omega} f(\omega) \qquad \omega \in \Omega \tag{2.14}$$

Optimisation problems can be categorised in two groups: unconstrained and constrained. The difference between them is that constrained problems have additional constrains which the objective function needs to satisfy. Based on the objective and constraints functions, it is possible to define:

- *Linear programming*: for problems with a linear objective function and linear equalities and inequalities constraints;
- *Quadratic programming*: for problem with a quadratic objective and linear equalities and inequalities constraints;
- *Non-linear programming*: for problem with non-linear objective function and constrains.

Demand estimation methods based on optimisation usually require response time or CPU utilisation datasets [LXMZ03, LWXZ06]. Optimisation is then used to estimate the demands in [KZT] with a load-dependent model using quadratic programming techniques. Differently, [Men08] uses different metrics, such as response time and arrival rate to estimate the system demands.

Despite their extensive validation on parameterization of models with isolated classes, none of the above methods have, to our knowledge, been validated in the presence of class-switching.

For the purpose of this work, we focus the attention on some of the most common non-linear optimisation algorithms such as Fmincon [BGN00, WMNO06], GlobalSearch (GS) [ULP+07], MultiStart (MS) and Genetic Algorithm (GA) [GH88, CGT97, Mit98]. Fmincon is a well known

optimisation algorithm based on the gradient-based method to find quickly the local minimum of the given objective function. However, if the function is too complex with several local minimum, this algorithm does not guarantee to return the global minimum of the function. For this reason, to make it more robust, Fmincon can be run multiple times from different starting points increasing the probability of finding a global minimum for the objective function.

The GlobalSearch and Multistart are more robust optimization algorithms that run several time the Fmincon algorithm. They just implement different strategies to identify the starting points for the Fmincon. For example, using the GlobalSearch, the initial starting points are chosen with the scatter search algorithm [Glo98] between the boundaries of the optimisation function. Then, it analyses each starting point and rejects those points which are unlikely to improve the best local minimum. On the other hand, the MultiStart algorithm chooses the starting points equally distributed inside the searching range and it runs all of them.

On the other hand, the Genetic Algorithm belongs to the evolutionary algorithms [Coe07] and, differently from the other considered algorithms, this algorithm is not based on Fmincon. The Genetic Algorithm starts its execution generating a population of possible configurations. Like in biological evolution, the algorithm selects randomly two configurations, called parents, to generate a new configuration, called child, that is a combination of the parents. All the children generated by the previous population are then becoming part of the next population and they are used as parents. The algorithm stops when one of the stopping criteria is satisfied (generation limit, time limit, constrains tolerance, etc.).

Differently from other optimisation algorithms and Fmincon, the genetic algorithm is able to work well with integer programming, so in problems where the possible values are restricted to integers. However, due to the randomness of the algorithm and since the next point to test is not deterministic (as in the other optimisation algorithms), it does not ensure that two executions of the same algorithm produce the same result.

# Chapter 3

# Apache Cassandra Queueing Network Model

## 3.1 Introduction

The growing importance of Big Data applications has led in recent years to a rapid growth of interest for NoSQL databases [LJ12]. In spite of their popularity, performance engineering for NoSQL databases is still in its early stages. In particular, NoSQL databases are commonly deployed on cloud platforms where hardware resources are rented, calling for dedicated provisioning methods in order to strike the right trade-off between costs and quality-of-service. Unfortunately, database performance can be challenging to manage in cloud environments, where it can be compromised by several factors, such as network contention and CPU multi-tenancy. For this reason, it is desirable to support engineers with analysis tools to assess risks associated to a target deployment.

Today, service providers tend to over-provision NoSQL databases, wasting resources and increasing the costs for their infrastructure. Finding the optimal configuration, in terms of number of nodes, number of replicas (*Replication Factor*) and data consistency (*Consistency Level*), is thus important and desirable, but still considered a research challenge. Performance prediction based on stochastic models can support sizing activities of this kind by allowing en-

gineers to easily compare alternative system configurations and predict performance and costs before deployment in the cloud.

One of the main challenges in modelling NoSQL databases is to properly account for the replication factors and data consistency levels. The goal of this chapter is to develop a model that addresses this need for NoSQL databases. Due to its popularity we analyse Apache Cassandra [LM10] and then we generalise the model to be used with other databases that use a similar data path. The data path of NoSQL databases is usually characterised by the data replication that is usually executed asynchronously, reducing response time despite a weaker data consistency level than traditional databases [Cat11]. In order to ensure consistency, read requests thus need to hit multiple replicas before the database system can respond to the client. This synchronisation can considerably affect system response time and, for this reason, in this work we focus our attention only on the read requests. Finding the optimal trade off between data consistency, costs and offered performance thus requires a quantitative approach that should assess all of these dimensions.

The proposed model is based on extended queueing networks consisting of fork-join elements, finite capacity regions, and class-switching. Since the model includes several features that are not readily amenable to analytical solution, we focus here on simulation-based assessment. Our performance model explicits both replication factor and consistency level, allowing engineers to compare alternative system setups.

We validate the proposed models against experiments based on the YCSB database benchmark [CST+10], under varying consistency levels and different infrastructures. Our results on a private cloud indicate that average error for the throughput is lower than 7% and lower than 10% for the response time, increasing prediction robustness compared to existing models. Also using the public cloud, the model is able to capture all the main database performance.

To enhance furthermore the accuracy of our model, especially in high load, we infer the demands for our system in multiple points and fit them in some mathematical functions. We then use these functions to parametrise our model setting the demands in correlation to the number of jobs in the system. In addition, we generalise our queueing networks model to represent a large

number of NoSQL databases with similar data path such as ScyllaDB. Moreover, in a case study, we demonstrate the utility of these models analysing the change in performance due to a different application behaviour.

The rest of the chapter is organised as follows. Section 3.2 summarises related work. Section 3.3 and Section 3.4 introduces the proposed queueing network and its parametrization, which are subsequently validated in Section 3.5. In Section 3.6 and 3.7, we demonstrate the applicability of our model to another NoSQL database, ScyllaDB, and propose a model-based analysis of Cassandra performance under quorum-based synchronisation. In the last section, the conclusions are discussed.

## 3.2 Related Work

Database performance modelling has been traditionally focused on relational databases [OK12]. With the increasing growth in popularity of NoSQL databases over the last ten years, research work in performance evaluation has been increasingly conducted also on these emerging database systems. Prior work has focused on comparing NoSQL systems in terms of read/write performance, scalability and flexibility [Cat11, LAV+15, BS15, KP14, KM17]. To the best of our knowledge only a limited number of works have been conducted on performance modelling of NoSQL databases, and available models specific to Cassandra are few [GGK+14, OP14]. The models in [GGK+14] and [OP14] use two different modelling techniques due to the difficulty to represent, in the same model, the synchronisation and the scheduling of the requests.

Gandini et al. present a high level queueing network model for Cassandra [GGK+14]. Each node of the distributed setup is described using two queues: one for the CPU and one for the disk. This model is able to capture throughput and latency for read and write requests. However, compared to our model, it does not take in consideration requests other than those executed locally and there is not synchronisation of the tasks between the nodes. Moreover, communication latency is not taken in consideration and there is no representation of the CPU tasks limit that Cassandra has.

Subsequently, [OP14] presents a model for simulating read requests. The authors use Queueing Petri Nets (QPN) [Bau93] in order to describe synchronisation. QPNs are an extension of coloured stochastic Petri nets, particularly useful when the model needs to explicit scheduling at queueing resources. Each node is represented by only CPU processes, described as a timed queue. Compared to the model in [GGK+14], the authors limit the number of simultaneously running queries for a node and distinguish between local and remote requests. However, no information about the disk and network latency is considered. In case of records of large size, the disk and network response time can considerably affect performance. The model in [OP14] is validated by predicting response time and throughput of different Cassandra configurations and appears to be the most accurate among existing performance models. However, for some particular configuration the error between real system and simulation can grow over 40%.

Compared to the previous models, our model is able to capture network and disk latencies, distinguishing the different type of requests (represented in the model as classes) and attributing different CPU demands to each class and to each phase of the query execution. These improvements deliver more accuracy on throughput and response time prediction. In addition, our model is able to represent different data replication factor and simulate the major consistency levels implemented in Cassandra.

Queueing Petri Nets have also been used in [Nie16] to predict and study Cassandra energy consumption. A study on how performance changes in relation to different data model structures has been conducted in [CKL15]. Chebotko et al. present a query-driven data modelling for Cassandra whose results are highly different from the one applied for the traditional databases. The paper also argues about the importance of the modelling consistency levels. The relation between performance and consistency level has been studied in several papers [BT11, ALS10, WFZ+11]. To achieve higher performance, Cassandra adopts eventual consistency instead of strong consistency. For this reason, to achieve strong consistency and meet the consistency levels required by the application without sacrificing the performance, variations of Cassandra database have been developed [CIAP12, GPM14]. Chihoub et al. presents a self-tuning mechanism to change at run-time query consistency levels, taking in consideration the system state. Differently, in [GPM14] the Cassandra's node architecture is modified to always

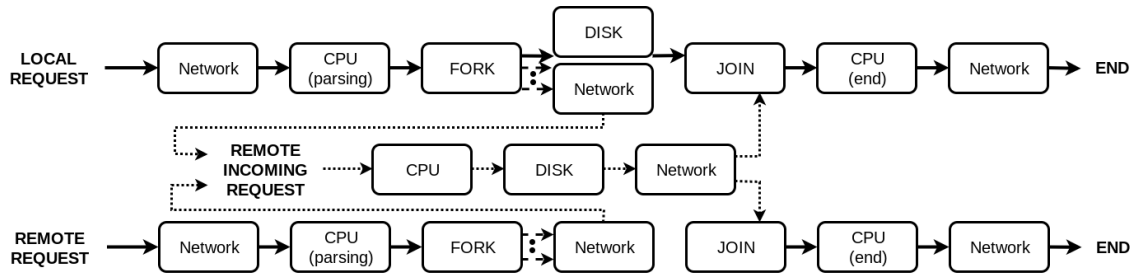



Figure 3.1: Local, Remote and Remote Incoming requests workflow.

achieve strong consistency of the data.

## 3.3 Cassandra Queueing Network Model

In this section, we present the Cassandra model used to predict database performance. The model is built using queueing network theory [LZGS84]. To represent and simulate our model, we use the JSIMgraph simulation environment provided with Java Modelling Tool (JMT) [BCS09].

As described in the background section, each node in Cassandra issues two main types of read requests: read data from the disk (*local request*) or retrieve data from another node (*remote request*). As part of a remote request, if a different node asks the data to the target node, the target node has to provide them (*remote incoming request*). Depending on the type of request that the node needs to process, the request follows different paths through the model. The different workflows are represented in Figure 3.1. In the figure, the boxes represent the node components used for a single node in our model. How the requests are forwarded are represented by the lines. Two different types of lines are shown: continuous lines indicate the connections inside the same node and dashed lines represent external node connections.

Figure 3.2 describes a single Cassandra node (identified as c1). The node is characterised by three queueing stations representing: network (c1_net), CPU (c1_cpu) and disk (c1_disk). For all the queues the scheduling policy used is non-preemptive first-come first-served (FCFS), with the exception of the CPUs for which processor sharing scheduling is used. To model all the functionalities that each node offers, other elements such as class-switches, a fork-join and a

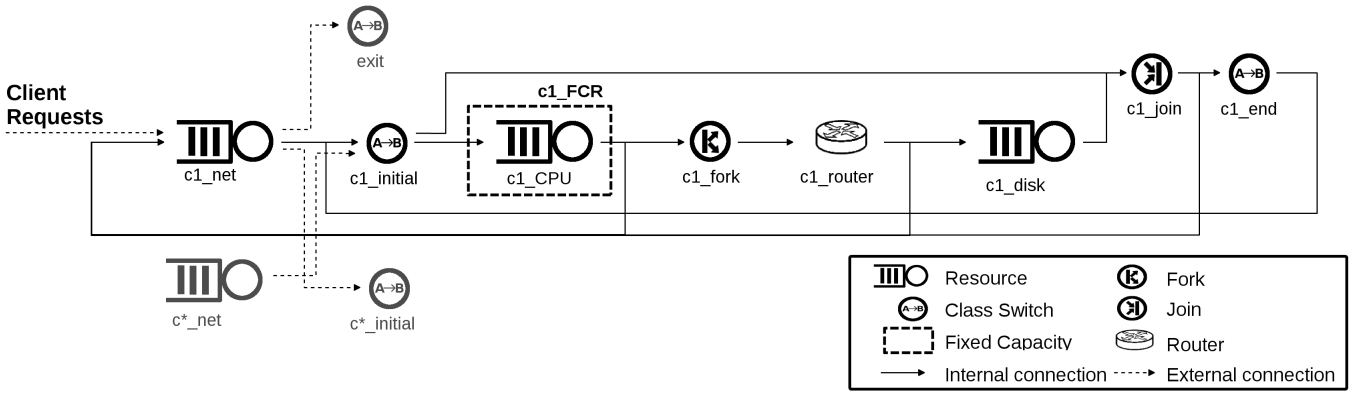



Figure 3.2: The model representation of a Cassandra node.

finite capacity region have been added to the model. In particular, class-switches are used to change the class of a request inside the model, the fork-join to manage multiple remote requests and finite capacity region to limit the number of simultaneous requests that the system can handle. This limit is static and set by default to 32 requests in the configuration file of each Cassandra node (parameter: *concurrent_reads*).

In addition, in order to model different types of requests, six different classes are defined in the model (described in Table 3.1). The number of classes is related to the number of nodes present in the system. Indeed, to maintain information about the origin of a request for each read-remote request, a read-remote-*ID* class for each node is created, where *ID* identifies a Cassandra node.

In our model, the initial requests received by the node from the client can be only local or remote and enter in the node via the network queue (c1_net element in Figure 3.2). Based on the type of request, different paths may be taken. Such paths are presented in the following sections. We also discuss model parametrization and workload characteristics.

### 3.3.1 Local Request

Local requests are queries for which the node stores data locally on the disk. In this case it does not need to contact any other node if the CL is ONE. Figure 3.1 shows the main steps for this type of request. The read-local request from the network queue goes directly to the CPU queue

| # | Class | Description |
|---|---|---|
| 1 | read-local | Local read request |
| 2 | read-local-end | Local request that needs to perform the end operations before to return to the client |
| 3 | read-remote-*ID* | Remote read request where the *ID* node is acting as proxy |
| 4 | read-remote-return | The returned remote incoming request performed by another node |
| 5 | read-remote | Remote read request |
| 6 | read-remote-end | Remote request that needs to perform the end operations before to return to the client |

Table 3.1: Classes description.

through the *c1_initial* class-switch. After it has been served, the request goes through the fork and router, reaching the disk. In the case of CL QUORUM or ALL, the fork generates as many remote requests as needed in order to satisfy the CL. When the disk finishes the execution of the request, it reaches the join (*c1_join*) where it waits until all the remote requests come back from all the other nodes. In case of CL ONE, the join fires the request immediately. Afterwards, the request class is changed in read-local-end inside the class-switch *c1_end* and it is forwarded to the CPU to perform the ends operations before it leaves the node through the network queue and the class-switch *exit*.

### 3.3.2 Remote Incoming Request

Remote Incoming requests (see the central row of Figure 3.1) are the data that a proxy node retrieves from the target node to complete a query. The data gathering process is defined similarly to local requests. Remote incoming requests enter in the node from the initial class-switch of the node c1. The class request arrived to the node c1 is one of the read-remote-*ID* classes. The request is executed by the CPU. The fork and router forward the request directly to the disk that retrieves the information and sends it to the join. The node network (*c1_net*) takes the request and based on the request class (read-remote-*ID*), forwards it back to the original node (*ID*).

### 3.3.3 Remote Request

Remote requests are the requests for which the node does not store the data locally. In this case, the node needs to act as a proxy and issue remote requests towards other nodes to complete it. Requests generated in this scenario are the remote incoming requests received from other nodes. Differently from the local requests, these queries never reach the local disk of the node. After it is served by the network queue, the request reaches the *c1_initial* class-switch in which the request changes class, becoming a read-remote-C1 request, and it continues execution into the CPU. As described in Table 3.1, this type of requests is characterised by two main CPU operations: parsing and end. When the request hits for the first time the CPU, it executes the parsing phase and after it is sent to the fork. Here, based on the CL applied to the queries, the fork generates as many requests as the CL needs. These requests are forwarded to the router and then redirected to the network queue. The requests are then randomly sent to some other nodes of the ring and executed as a remote incoming request by another node.

When the read-remote-C1 requests come back from the remote nodes, they change class in read-remote-return (inside *c1_initial*) and they are forwarded to the join. When all the generated requests come back, the join fires generating a single read-remote-end request that is sent to the CPU via the initial class-switch. Inside the CPU a read-remote-end request is processed to perform the end operations. Finally, the request exits from the node through the network queue (*c1_net*) and is sent back to the client through the *exit* class-switch.

### 3.3.4 Workload

Another important component of our model is represented by the workload generator. To the best of our knowledge, two benchmarks are available to measure the performance of a Cassandra cluster: cassandra-stress tool and YCSB. Cassandra-stress tool is developed by Datastack and shipped with Cassandra. On the other hand, Yahoo! Cloud System Benchmark (YCSB) [CST+10] was initially developed by Yahoo! and then released under the Apache License.

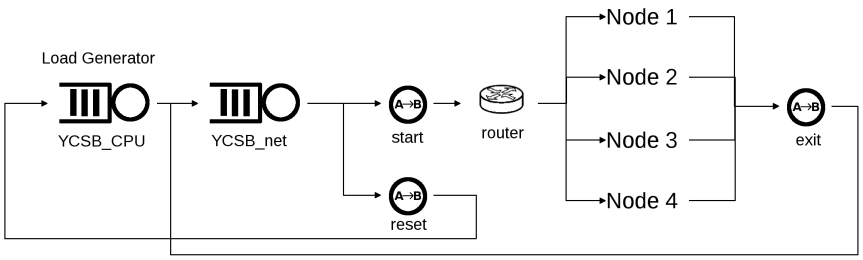



Figure 3.3: Cassandra model overview at high level of granularity.

Differently from cassandra-stress tool, YCSB is a benchmark tool that supports many different types of applications and databases and new features or support for new applications are added regularly. For this reason, it is a quite popular framework used in industry and academia [ABF14, MAS19, NL18]. The two tools provide similar functionality and are able both to generate write-only, read-only and mixed workloads using different distributions. However, for the type of experiments that we need to run, YCSB gives us more flexibility especially displaying the partial performance results of the cluster during the experiments. So, for this reason, for our experiments through this chapter and the whole thesis, we decided to use the Yahoo! Cloud System Benchmark (YCSB) [CST+10].

Figure 3.3 provides an overview of the system, including the YSCB workload generator. In order to approximate better the performance of our model, we have decided to model it with two queues that represent the CPU and the network running the workload generator machine. YCSB CPU queue manages only read-local class requests. The new requests generated by the CPU are sent directly to the *start* class-switch through the network queue. This class-switch changes in read-remote some of the requests, according to the replication factor and CL applied to the system. The proportion of read-remote requests is equal to $(N - RF)/N$. Then, read-local and read-remote requests are forwarded to a router that sends them randomly to the nodes inside the ring.

When the requests have been executed by the node, the network queue of the node sends the response to the *exit* class-switch which changes class to read-remote-end. In the end, before being received by the YCSB CPU, the request is sent to the YCSB network and to a class-switch to reset the class in read-local. When the query comes back to YCSB CPU queue, the request

is considered completed and a new query is generated by the YCSB workload generator.

## 3.4 Models Parametrization

To parameterize the model, we need to estimate disk, network capacity and CPU demands for YCSB and Cassandra nodes. To gather this information, we set up a Cassandra ring with four nodes and an additional machine for YCSB workload generator. Regarding network latency, two different requests are taken in consideration: request (or query) and response. They differ in size of data that they are carrying. We have observed that a normal request size is around 128 bytes. Differently, the response packet is characterised by the size of the data asked. To calculate the network demand, we multiply the packet size with the bandwidth information gathered from the Linux tool *iperf* [ESn16]. Similarly, we have estimated disk demand using the data size and the Linux tool *hdparm* [Lor16].

For the CPU demand estimation, three different approaches can be used to calculate them. The first two methods gather the information from the network packets exchanged between the nodes and the clients. The main difference between these two methods is on the number of multiple clients that can be used in the system during the parametrization phase and the ability to read the network packet's content. The first approach limits the number of concurrent requests to one but it does not require to read the content of the transmission. On the other hand, the second method does not introduce any limitation about the number of clients that are querying the system but it needs to be capable to read the packets' content to identify the requests and calculate the demands.

Differently, the third approach samples some queries and it analyses all the events generated by the database to complete them. We observe similar demands obtained with the first and third methods. However, the third method allows the user to have more clients in the system and to not have access to the operating system. In the following sections, we are going to illustrate all the three approaches.

### 3.4.1 Single client network monitoring

The first approach observes the network traffic exchanged between the nodes and the workload generator (YCSB). To calculate the demands, the algorithm requires only the query timestamps and response times. In fact, this algorithm does not analyse the content of the system but it looks only to the packet exchanged between the cluster. Analysing the IP address of different packets, we categorise three types of read requests. For the demand estimation of the different classes we took in consideration the timestamps packets of:

- Local request: the packets arrived from and departed to YCSB IP address.
- Remote incoming request: the packets arrived from and departed to each other node IP address.
- Remote request (parsing phase): from when the packet arrives from YCSB to the last packet departure directed to another node.
- Remote request (end phase): from when the first packet comes back from a node to YCSB response.

In case of local requests with CL QUORUM or ALL, the demand is calculated as a remote request and we differentiate between the two requests based on the number of remote-incoming requests generated by the request coordinator. To obtain the final demand for each class, we process the gathered data using the Complete Information algorithm [PPSC13]. In the same way, we also estimate CPU demand of the YCSB workload generator. The timestamps taken in consideration in this case are, simply, the packets that the server sent to and received from a node. To avoid the simultaneity of multiple requests, all the demand estimation with this method are taken with a single job in the system.

### 3.4.2 Multi clients network monitoring

This method is an extension of the method illustrated in the previous section. Differently from the previous one, this method provides the visibility of the content of the packets transmitted to calculate the system CPU demands. The message content is used to identify the different requests that are executed on the system and, with the source and destination IP addresses, the stage of each request. To analyse the traffic, we sniff all the network connections using tcpdump Linux tool [Tcp18] and analyse them offline. However, to be able to see the content of the file in clear text, we disable to network compression algorithm and disable the encryption if it is used or to be in possess of the SSL certificates used.

The recorded traffic is then processed by a Python script that we develop to reconstruct the execution of each query. Since in this case we can observe the content of the transmission, we can identify the primary key of the request and group all the packets for that query together. Then, we analyse each group of packets as described in the previous section and so we divide the query into the different classes. Analysing the timestamps, we gather the different information to be used with Complete Information algorithm [PPSC13] or any other algorithms.

### 3.4.3 Cassandra tracing tool

This approach leverage on a completely different mechanism compared to the previous two methods and involves the utilisation of the *tracing* troubleshooting tool. This tool is shipped with Cassandra and it is able to profile the internal operations that are executed by Cassandra to complete a query. Once enabled, it starts to record all the steps that the NoSQL database performs to complete the query with relative timestamps and the node on which it has been executed. However, this tool can impact severely on the database performance since several writes needs to be executed by the tool to store the information on the database.

In order to prevent performance degradation, this tool allows the database administrator to set a ratio of queries to record. Since, the estimation algorithms that we considered require to have all the requests executed over a period of time, we cannot use this feature and we need to

record all the queries. For this reason, even if this method allow us to parametrize the system having multiple users and without touching the underground platform, in our experiments, we evaluate this method with maximum 10 clients at the same time.

To estimate the CPU demands using the tracing tool, we divide the computation executed by each node to complete the query identifying the different stages of the request based on the description of the task. For each computation segment, we calculate the time that the system spends to complete the computation looking the initial and ending timestamps. In case of a local request with a CL of ONE, since the execution is conducted on only one node, the CPU demand for this class is the total time the node spends to complete the query without considering the disk read operation.

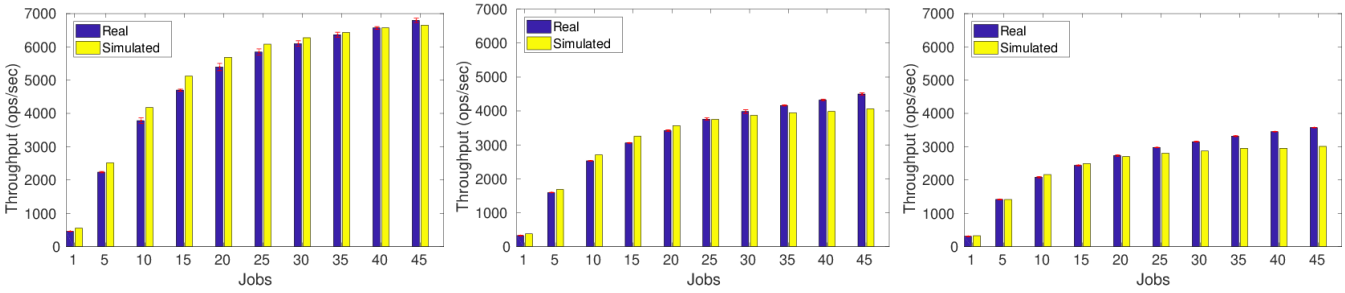
Differently, for a local request with CL greater than ONE or for a remote request, the contribution of each node is calculated considering:

- Local or Remote request (parsing phase): from the beginning of the tracing execution of the query to when the system create and send the request to the other nodes.

- Local or Remote request (end phase): from when the system received all the Remote Incoming requests containing the data from the other nodes to the end of the execution of the query.

- Remote incoming request: from when the node receives the remote incoming request up to its completion. This CPU demand does not take in consideration the time spent for the reading operation from disk.

As for the other methods, the final CPU demands for each class are obtained using one of the inference algorithms.




(a) ONE (b) QUORUM (c) ALL

Figure 3.4: Throughput comparison between Cassandra system and our model.

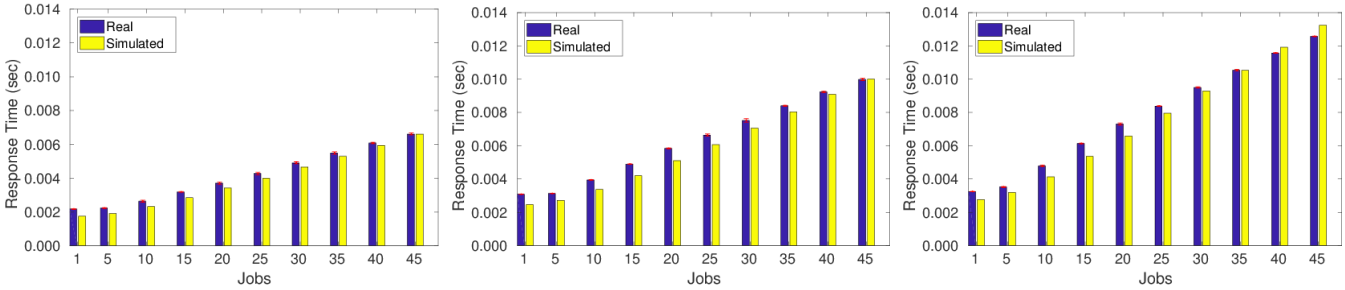
## 3.5 Model validation

Here we illustrate the evaluation of our model on different platforms, using different hardware resources and different architectures. In all the analysed cases, we demonstrate that the model is able to predict with accuracy the system performance. However, we notice that when the system is heavily utilised, the performance of our model slightly decreases due to several caching mechanisms that the database uses to reduce the latency of the system. For this reason, we infer the demand of our system on multiple points and then we fit a function for each class using these points. Using the parametrized functions, we obtain the final set of demands to set on the model. In this way, we obtain better performance prediction for our model.

Furthermore, we then demonstrate that our model has the flexibility and capability to represent also other NoSQL databases that present a similar system behaviour on how they process the queries. In the end, we present a case study on which we demonstrate the utility of models like this one when different application behaviour need to be tested, saving a considerable amount of time and money to develop them.

### 3.5.1 Validation on private cloud

We start the validation of our model using the private cloud. We decided to start with this testbed because we can control better the noise generated by the neighbour machines and produce better CPU demands. To validate our queueing network model, we set up a Cassandra cluster composed of four nodes with a replication factor of 3. The private cloud manager we




(a) ONE (b) QUORUM (c) ALL

Figure 3.5: Response time comparison between Cassandra system and our model.

Table 3.2: CPU demands for different classes.

| **Class** | **ONE** | **QUORUM** | **ALL** |
|---|---|---|---|
| read-local | 0.8028 | 0.4078 | 0.5 |
| read-remote | 0.374 | 0.5712 | 0.6 |
| read-remote-*ID* | 0.6309 | 0.6423 | 0.6 |
| read-local-end | 0 | 0.4968 | 0.5 |
| read-remote-end | 0.4994 | 0.8917 | 1 |
| read-remote-return | n/a | n/a | n/a |

Table 3.3: Different component demands.

| **Component** | **Demand value** |
|---|---|
| Network Request | 0.01333 |
| Network Data | 0.1333 |
| Disk | 0.0679 |
| YCSB CPU | 0.428 |

use is OpenNebula which uses KVM as hypervisor to run the machines. The machines used as Cassandra nodes have 2 CPUs and 4 GBs of memory. Each machine has 2 disks attached: a 20 GBs, where Cassandra commits logs and saved cache files are stored, the second of 100 GBs to store only database data. The operating system is Debian 8 with Sun Java 1.8 and Cassandra version 2.2.4. For the purpose of these experiments, we disable any Cassandra caching mechanisms and we maintain the default settings of the operating system. We also ensure that the data stored in the database does not fit all in memory otherwise this can change the database behaviour. Since we are using the cloud, we also paid attention to the CPU steeling value to be zero or very close to it. To avoid interfering with the performance of Cassandra nodes, the workload generator YCSB is installed on a separate virtual machine with 4 CPUs and 4 GBs of memory.

The database is set to equally distribute the key range between all the nodes in the ring. For the workload generator, we use the default parameters that come with YCSB. Each row of the database is composed of 10 fields, each filled with 100B of data. The total data size for each record is 1KB, excluding the identification key. The keyspace is filled with 15 million of records. These parameters can be easily changed within the YCSB workload generator to

represent more realistic user scenario of the system under test. We assume that the data size inside the table can variate with a limited level of degrees. In this situation, setting the correct number of columns and using the average data size for each column, the model can predict with accuracy the cluster performance. On the other hand, if a significant diversity of data size is stored inside the table, higher percentage of error can be observed in the model. This error can be reduced identifying the ratio of which these requests occur on the system and creating, inside the model, an additional class where more specific demands for these kinds of queries can be specified.

To emulate the different number of simultaneous clients in the system (also called jobs), the same number of threads are set in YCSB. Each experiment on the real system is repeated three times and it has the duration of 30 minutes with 5 minutes of warm-up before starting to record.

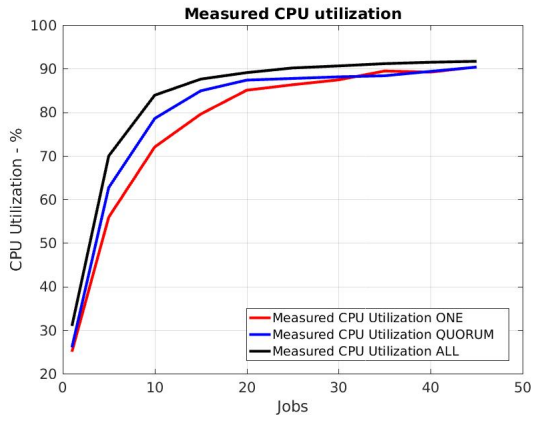



Figure 3.6: Cassandra CPU utilisation.

| CL | Throughput | Response time |
|---|---|---|
| ONE | 6.79 % | 8.17 % |
| QUORUM | 6.41 % | 9.59 % |
| ALL | 6.98 % | 7.64 % |

Table 3.4: Model relative error.

The evaluation compares the throughput and response time of the real system with the results obtained through the simulation of our model using JMT tool. The demand values used for the model simulation are reported in Tables 3.2 and 3.3. For this set of experiments, the CPU demands are calculated sniffing the network with a single client on the system. Figure 3.4 and Figure 3.5 shows the average throughput values recorded by YCSB at the end of the execution of each experiment. The figure also shows the values range recorded for the same experiment from the real system.

Before comparing the result of the two approaches, we can analyse the effects that different CLs

| Algorithm | Local Pars | Local End | Remote Pars | Remote End | Remote-Incoming |
|---|---|---|---|---|---|
| CI | 0.00004801 | 0.00010055 | 0.00005501 | 0.00055917 | 0.00036023 |
| QMLE | 0.00004456 | 0.00030543 | 0.00004569 | 0.00066055 | 0.00038402 |

Table 3.5: Demands with 50 clients and CL ALL

Table 3.6: Microsoft Azure details.

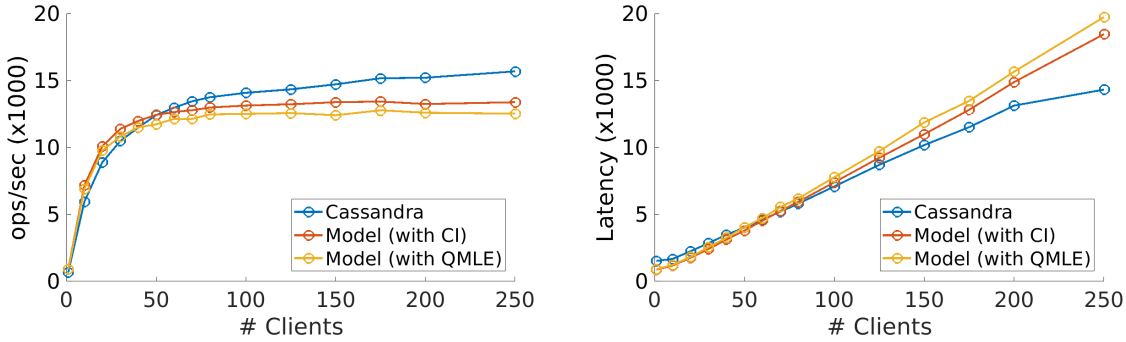
| | |
|---|---|
| Cassandra machines | 4 |
| YCSB machine | 1 |
| Machine type | Standard_D2s_v3 |
| CPU Frequency | 2.4 GHz |
| vCPU | 2 |
| Memory | 8 |
| Disk | 100GB |
| Max IOPS | 4000 |
| Storage type | SSD (premium) |

Table 3.7: Demands with 60 clients and CL ONE

| Alg. | Local | Remote | Remote-Inc. |
|---|---|---|---|
| CI | 0.00034926 | 0.00068336 | 0.00013746 |
| QMLE | 0.00043431 | 0.00065454 | 0.00014813 |
| ERPS | 0.00048365 | 0.0006546 | 0.00017278 |

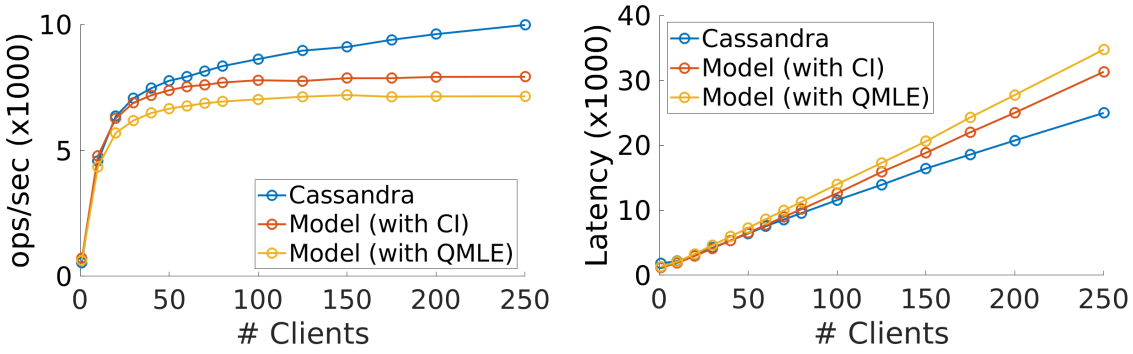
have on the real system. Using ONE as CL with 45 jobs, or 45 simultaneous client requests, the system is able to serve around 6800 requests per second. Compared to ONE, the throughput of the system is reduced more than 40% and up to 60% when QUORUM and ALL CL are respectively used. In fact, increasing the CL, the CPU utilisation of each node grows faster like the graph reports in Figure 3.6. In addition, the response time also changes considerably, increasing linearly the average time that the clients wait for query completion.

Comparing the results obtained from the real system and from our queueing model, it is possible to notice that our model captures all major changes in the database. As shown in Figure 3.4, the model is able to approximate with a very small error the throughput of the real system, as the number of jobs and CLs vary. As presented in Table 3.4, the average throughput relative error is below 7% for all analysed CLs. Using our model, we also estimate the response time of the queries for the same experiments. The results are reported in Figure 3.5. Even in this case, the model is able to predict with good accuracy the real system performance. The average response time error, as reported in Table 3.4, is below the 10%.




(a) Throughput (b) Response time

Figure 3.7: Model performance on public cloud with CL ONE




(a) Throughput (b) Response time

Figure 3.8: Model performance on public cloud with CL ALL

### 3.5.2 Evaluation on public cloud

In this section, we test our model performance on the cloud. Since the underline infrastructure is managed by the cloud provider, we do not have control on the other machines around ours and this can lead to serious noise during the execution of the experiments and the parametrization of the model. In addition, in this section, we compare the performance of three demand estimation algorithms CI, QMLE and ERPS. When we started the investigation we also included the UBR algorithm on the list of inference algorithms. However, due to *collinearity*, a known effect of this regression model, we were not able to use it. In this case, the collinearity is caused by the constant rate of local and remote requests. After the performance evaluation of the model in the cloud, we improve its performance prediction fitting the demands in a function. This method helps to reduce considerably the error of the model especially when it is heavily loaded.

| Algorithm | Throughput | Response Time |
|---|---|---|
| CI | 10.87 % | 13.55 % |
| QMLE | 12.59 % | 15.46 % |

Table 3.8: Overall model performance prediction relative error with CL ONE.

| Algorithm | Throughput | Response Time |
|---|---|---|
| CI | 10.56 % | 11.66 % |
| QMLE | 17.16 % | 19.52 % |

Table 3.9: Overall model performance prediction relative error with CL ALL.

The cloud provider used our experiments is Microsoft Azure on which we deployed four Cassandra nodes and one YCSB machine. The replication factor used by Cassandra for these experiments is 2. The hardware characteristics are reported in Table 3.6. The database has been loaded with 30 million records of 1KB each divided across 10 fields as in the previous experiments. The workload generator YCSB is set to use a read only profile as previously done. Due to the resource contention problem related to the use of a public cloud provider, during our experiments we monitored the CPU stealing value to ensure that is zero of negligible.

Since the hardware characteristics of the machine used for this set of experiments performs better than of the previous one, we have the possibility to test the system with more clients querying at the same time the database. As we can see from Figure 3.7a Cassandra reaches the maximum throughput around 100 clients in the system. Afterwards, the response time increases linearly while the throughput remains almost the same. This effect is known as performance knee of a system and it is presented when the system is saturated. Indeed, with 100 clients, each node in the system has a CPU utilisation of over the 90%. The small increment of throughput that the system has with more than 100 clients, is represented by some caching applied by the operating system. However, due to the high CPU utilization, we do not advise to rely on the model or run a production system where the CPU is completely saturated. In addition, episodes of high contention of the resources and noise are likely to occur since we are using a public cloud provider.

To calculate the demands to parametrize the model, we run the system with 60 clients at the same time for the CL ONE and 50 clients for the CL ALL. In both cases, we record all the network packets exchanged by the nodes and, using the second method presented in the previous section, we divided them in the different model classes. We decided to estimate the nodes demands using the CI, the QMLE and ERPS [WCKN16, PPSC13, SCBK15]. These

are quite popular advanced estimation algorithms with different characteristics. In particular, the Complete Information (CI) is a scheduling-aware estimation algorithm for multi-threaded applications that uses linear regression and maximum likelihood estimators. The algorithm uses information about the number of requests and their execution to infer the demands of each class. On the other hand, Queue-length Maximum Likelihood Estimator (QMLE) uses maximum likelihood techniques to infer the demands of load-independent and load-dependent resources in systems with parallelism constraints. To infer the demands, the algorithm requires the queue-length at the different resources that can be easily collected from a real system. In addition, this algorithm is also able to provide a confidence interval for each demand. Differently, ERPS is a regression-based methodology that approximate CPU demands. This is an advanced regression-based solution because it is able to reduce the level of uncertainty (or noise) present in the recorded traces compacting workload information. To the best of our knowledge, these are only inference algorithms that support multi-threaded applications. To calculate the CPU demands to set in the model we use the Filling the Gap (FG) tools [WPC15] which support all the inference algorithms described above.

The results gained by these algorithms are reported in Table 3.7 for CL ONE and 3.5 for consistency ALL. From Table 3.7, we can notice that the CI presents values for the local and remote-incoming requests lower than the one presented by the QMLE. On the other hand, the ERPS values are always higher than the QMLE. This lead the model with ERPS to perform worse than with the other two algorithms. For this reason, we do not consider the ERPS on further experiments. A different scenario is shown with the Consistency Level ALL (Table 3.5) where for the local end, remote pars and remote end classes, the demands are very different.

The model throughput and response time are presented in Figure 3.7a and Figure 3.7b. Observing the figures, it is evident that the model is able to capture the trend of the real system and predicts with a low relative error the system performance. Moreover, the demand estimated with the CI algorithm are performing slightly better than the one produced by the QMLE. Indeed, as reported in Table 3.8, using the CI demands, we obtain an error below 11%. Furthermore, a considerable portion of error is obtained when the server is heavily utilised with 150 clients or more. In fact, with the QMLE and 150 clients in the system, the model prediction

error for the throughput and latency are over 20%.

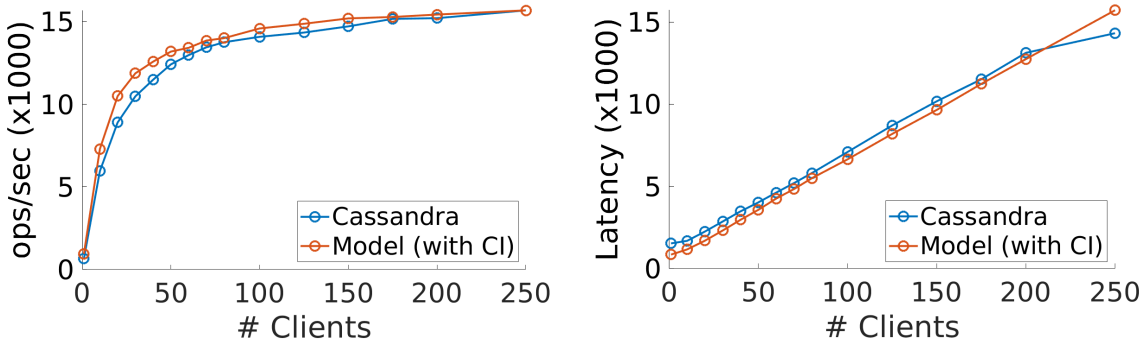
Also, with consistency level ALL, the model applied on the public cloud performs very well. In these experiments, the CI achieves even a lower percentage of error compared to the one registered with consistency level ONE. On the other hand, the QMLE presents a slightly higher error than before, as reported in Table 3.9. The throughput and response time prediction for the different number of clients are presented in Figure 3.8a and Figure 3.8b. With both the consistency level analysed, the model decreases its accuracy when the system becomes fully utilised. For this reason, in the next section, we improve the performance of the model estimating of the demand in several points and fitting a mathematical function to be used to estimate the model demands.

### 3.5.3 Fitting demands function

Using the same public cloud test environment, we tried to enhance the performance of the model, fitting the demands into some mathematical functions that we consequently used to parametrize the model. To fit the function, we record several network communications with a different number of clients in the system. Then, we parse the communications to extract all the necessary information to run the CI algorithm and gather the different demands for each class. We try to interpolate the data using the linear, cubical, polynomial, and exponential functions and we select the algorithm that returns the lower Root Mean Square Error value (RMSE). We notice that, in the Cassandra case, two functions can interpolate better the demands and this is also related to the Consistency Level used. In particular, the a linear function is used for consistency level ONE and exponential with ALL. We decided to not introduce any other values a part from the estimated demands in our fitting model to have a clear comparison between the model that use the fitting technique and the one without. Introducing other metrics such as CPU utilization or requests response time in the fitting model can be useful in some situations, but can also move away form the real demand value on that point especially when the system is highly loaded. We take in consideration the case where 4 and 7 points are used to fit these functions.

| Class | CL ONE | | CL ALL | | | |
|---|---|---|---|---|---|---|
| | p1 | p2 | a | b | c | d |
| Local Pars | 1.752e-08 | 7.751e-07 | 0.0001172 | 0.001907 | -0.0004428 | -0.1341 |
| Local End | n/a | n/a | 0.0001303 | 0.001026 | -0.0005845 | -0.1514 |
| Remote Pars | 2.086e-08 | 3.683e-07 | 0.0001457 | 0.00155 | -0.0004739 | -0.119 |
| Remote End | -1.101e-07 | 0.0005908 | 0.0002198 | 0.001325 | -0.0003855 | -0.05859 |
| Remote Incoming | -6.092e-07 | 0.0005283 | 0.000655 | -0.04182 | 0.0003963 | -0.001356 |

Table 3.10: Values for 4 points function with CL ONE and ALL




(a) Throughput (b) Response time

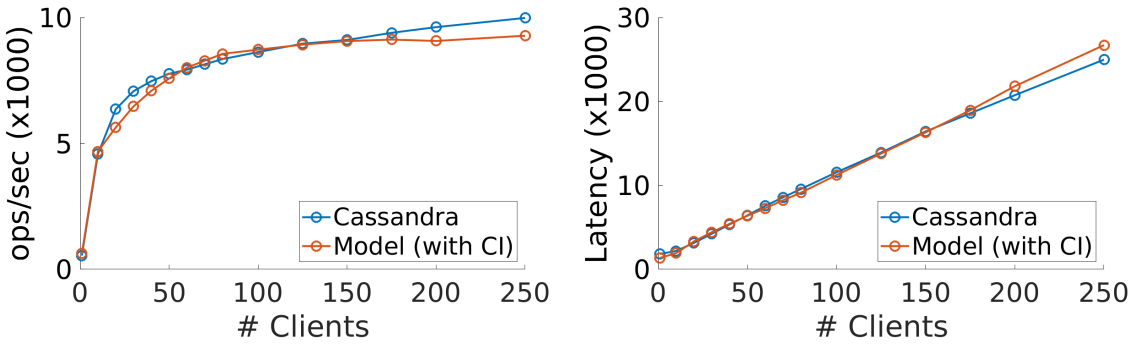
Figure 3.9: Model performance with demand fittings with 4 points and CL ONE

We started our analysis recording the demands with 10,30,100,250 clients and a consistency level of ONE. We then fit the points on a linear function, like $f(x) = p1 * x + p2$. Table 3.10 reports the values obtained from Matlab R2017b to parametrize the linear functions.

The performance of the model using the 4 points function are presented in Figure 3.9a and Figure 3.9b. It is possible to notice that the model is able to perform considerably better in the prediction of the throughput with an average relative error of 8.96% compared to 10.87% recorded without using any fitting technique. Regarding the response time, no significant change has been registered.

We applied the same technique also with consistency level ALL. However, in this case, we use an exponential function like $f(x) = a * exp(b * x) + c * exp(d * x)$ since the Root Mean Square Error (RMSE) returns a better value. Using this fitting, we reduce significantly the average error of the model to 4.57% for the throughput and 5.40% for the response time. The accuracy of the model in this case it is also visible in the Figure 3.10a and 3.10b.

We further investigate the possibility to use more points to fit the demand function. In the




(a) Throughput (b) Response Time

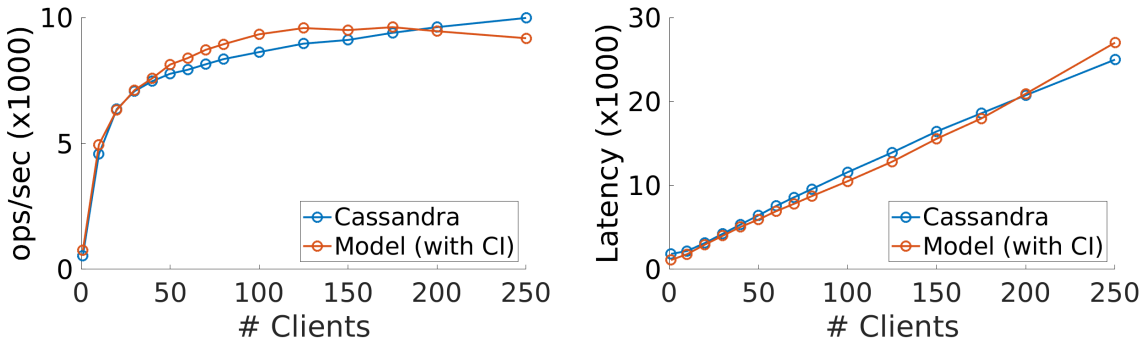
Figure 3.10: Model performance with fitting with 4 points and CL ALL

| Class | a | b | c | d |
|---|---|---|---|---|
| Local Pars | 0.00009719 | 0.002647 | -0.0001045 | -0.04518 |
| Local End | 0.0000969 | 0.002201 | -0.00009772 | -0.03115 |
| Remote Pars | 0.0001317 | 0.001954 | -0.000136 | -0.03382 |
| Remote End | 0.0001611 | 0.002566 | -0.0001625 | -0.03024 |
| Remote Incoming | 0.007972 | -0.3288 | 0.0005403 | -0.002972 |

Table 3.11: Values for 7 point function with CL ALL

case, we estimate the demands in 7 points, with 1,10,30,50,100,175,250 clients, and with a consistency level ALL. We fit the class demands in an exponential function and we report the value in Table 3.11.

The performance of the model using the 7 points function are presented in Figure 3.11a and Figure 3.11b. It is possible to notice that the model is able to perform better in the prediction of the response time with an average relative error of 9.52% compare to 11.66% that we recorded without using any fitting technique. Also, the throughput error is decreased with an average of 7.03%. However, the performance of the model in this case are lower than the one presented with 4 points function. Ideally, we are expecting that adding more points to the fitting model, this would enhance the precision of the queueing network model as well. However, we noticed that the fitting for the Local End class presents some overfitting effect recorded using the r-squared measure. We attribute the additional error of the queuing network model performance compared to the model using 4 points to the overfitting of this class.




(a) Throughput (b) Response Time

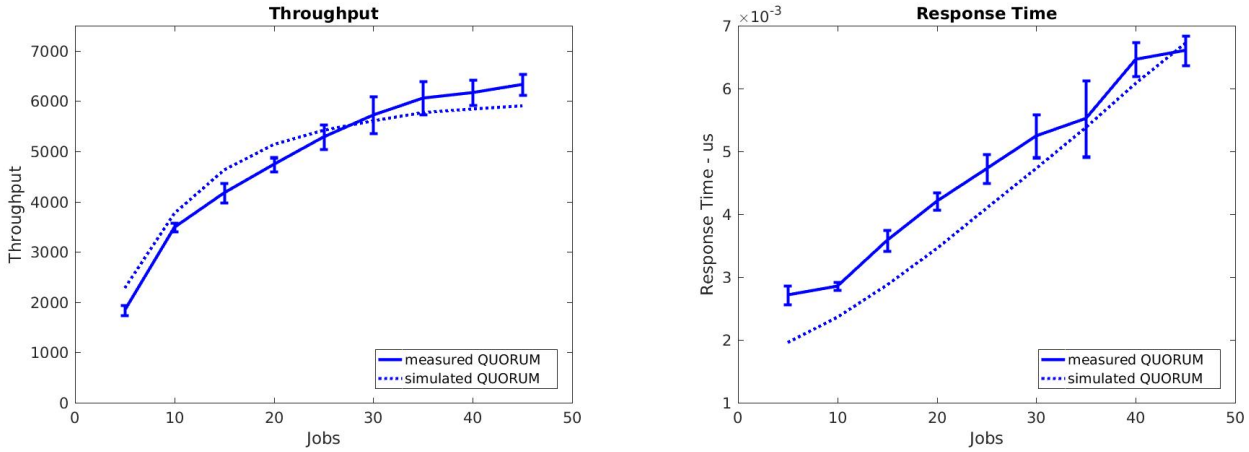
Figure 3.11: Model performance with fitting with 7 points and CL ALL

## 3.6 Case Study: applicability of our model to other NoSQL databases

As part of our evaluation, we consider the possibility to apply our out-of-the-box model to other NoSQL databases. Due to the similar system architecture and behaviour of the queries execution, we decided to apply the model to ScyllaDB.

ScyllaDB is an Apache Cassandra compatible column store database that aims to optimise the performance of each node participating to the database. This increment of performance is obtained using a different programming language (C++) and a completely new software architecture optimised for modern hardware resources. Typically, NoSQL datastores run on a single Java Virtual Machine. To reduce the idle periods generated by the usage of the Garbage Collector, which tend to increase the latency of the system, these applications usually use page cache and complex memory allocation strategies. To avoid these expensive locking mechanisms, ScyllaDB uses an engine for each CPU core of the system, which operates with a low degree of synchronisation. This reflects on the database with a lower usage of the hardware resources and an increment of database performance.

To gather performance data for ScyllaDB, we deploy the same Cassandra configuration used for the private cloud. The ScyllaDB deployment is composed by four machines, with a replication factor of three and 1KB of data for each record stored. Due to the optimised usage of the recent hardware features, ScyllaDB requires the presence of specific features of the CPU on




(a) Throughput (b) Response time

Figure 3.12: Model performance with ScyllaDB.

the VM. For this reason, ScyllaDB VMs are deployed using a different CPU architecture that supports the SSE4.2 CPU flag instructions [Int16b]. To accomplish ScyllaDB requirements, we force the hypervisor to create virtual machines with the Intel Nehalem CPU microarchitecture [Int16a]. The usage of this microarchitecture also brings more benefits to the hardware resources allocated to the VM thanks to a larger bandwidth for the network and disk.

To adapt the model for ScyllaDB, we also need to remove the fixed capacity region set on the CPU because ScyllaDB does not limit the number of simultaneous queries that each node can handle. To parametrize our model, we:

- Estimate the network and disk bandwidth using the Linux tools hdparm and iperf. The two bandwidths have been multiplied with the average size of the data stored in the database or query network packet size.

- Estimate the YCSB CPU demand analysing the network packet exchanged between the client and a node. We measure the latencies that the client takes to generate a new request when a complete one is received.

- Estimate the node CPU demand for local parsing operations sniffing the network communication of a node with a single client. The packets are then analysed to measure the latencies from the instant which a client query arrives to the node to the one in which

the node sends the remote incoming request to another machine. This estimation process is the same also for the remote parsing class. These two classes are distinguished by the number of requests that the node generates. The local requests generate one request less than the remote since a copy of the asking data is stored on the local disk of the node.

- Estimate the node CPU demand for local end operations sniffing the network communication of a node. The packets are then analysed to measure the time that a node takes from the instant which the asking data arrives back to the node until when the response to the client is sent. In case of CL ONE the demand for this class is set to zero. The same process is used for the CPU estimation demand of the remote end class.

- Estimate the node CPU demand for remote incoming requests sniffing the network communication of a node. The packets are then analysed to measure the time that a node takes from when a data request comes to the server to when the request is completed and it is sent back to the asking node.

All the CPU demands are calculated with a single job in the system and the collected latencies are processed with the Complete Information algorithm [PPSC13] to obtain the final demand for each class. The model is parameterised with the calculated demands for all the classes and, according to the real implementation, the same number of CPU cores is set for each node.

ScyllaDB results, measured and simulated, are shown in the Figure 3.12. They represent, respectively, the throughput and the response time of the system. For this case study, only the QUORUM case is taken in consideration. Comparing ScyllaDB with Cassandra QUORUM results, it is possible to observe that ScyllaDB is able to achieve better performance. We observed that, differently from the Cassandra scenario, at high load, the workload generator network represents the bottleneck of this set up. The results provided by the simulation tool are close to the measured one and the average relative error is 8% for the throughput and 12.9% for the response time. With this case study, we demonstrate the versatility of our model and the possibility to apply this model to other NoSQL database with minimal changes.

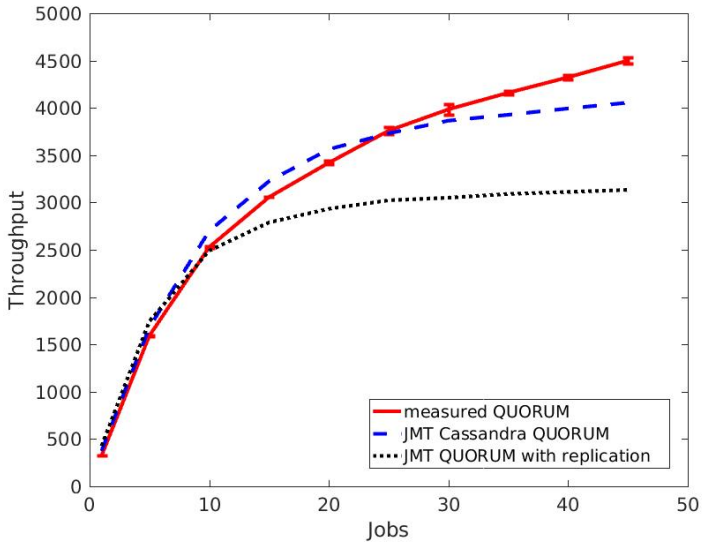



Figure 3.13: Throughput impact of query replication.

## 3.7 A What-If Scenario: the Impact of Query Replication in Cassandra

We conclude by illustrating an application of the proposed model. Recent research have highlighted the potential advantages of replicating requests in systems in order to consistently achieve low latency [QP15]. We here consider a variant of the proposed model to assess the potential impact of query replication on Cassandra.

In the default configuration, when the CL is set to QUORUM, Cassandra generates a fixed number of remote requests. The number of remote requests depends on the RF applied to the database and is equal to $\lfloor \frac{RF}{2} \rfloor + 1$. We now consider a variant of this approach that generates $RF$ requests, but where only the first $\lfloor \frac{RF}{2} \rfloor + 1$ responses need to be synchronised before completing the query. In this way, the system neither needs to apply any decision on which nodes to send the requests and does notit need to keep track of node performance. Clearly, we expect a trade-off between the enhancement of performance obtained from the higher parallelism and the heightened contention placed on physical resources.

In order to assess this scenario, we have devised an extension of the JMT queueing simulator which allows to synchronise a subset of requests at join nodes. Requests that reach a join

after this has completed the synchronisation are ignored in the computation of performance metrics. This extension is now integrated in the stable release of JMT version 0.9.3 under the name of *Quorum Join Strategy*. Figure 3.13 illustrates simulation results. We here compare real Cassandra measurements, with the proposed Cassandra QUORUM mechanism proposed in the previous sections, and the real QUORUM that refers to the replication mechanism.

We notice that with a very light load, the throughput under replication is nearly identical to the one without replication. However, at higher loads Cassandra is expected to suffer on average a performance loss compared to the system without replication. This simulation results seem to indicate that increasing replication levels might not be beneficial in a system like Cassandra. While experimental evidence would be needed to corroborate this prediction, this example provides intuition on the ease of performance analysis that comes with the proposed models.

## 3.8 Summary and Conclusion

In this chapter, we propose a novel queueing network model for the Apache Cassandra and other NoSQL databases. We evaluate our model with different hardware resources, different architectures and on private and public cloud. Simulation-based analysis shows that our model is able to predict with an error below the 10% the throughput and response time of the system under different consistency levels and number of concurrent requests. The proposed model appears generally more accurate than existing models in the literature that also consider the YCSB benchmark for validation purposes. Furthermore, we increase the accuracy of the model measuring the demands on few points and fit them on a mathematical function. In this way, the model is able to achieve even lower error in the prediction of the throughput and response time of the real system.

As part of this chapter, we also demonstrate that, with few adjustments, our model could be used to predict the performance of other NoSQL databases. Moreover, we illustrate the utility of such model to quickly develop and analyse the performance of the system if, for example, the data request is processed by all the nodes that stores a copy of the data.

# Chapter 4

# Partition-Aware Autoscaling for the Cassandra NoSQL Database

## 4.1 Introduction

NoSQL databases offer the ability to store large quantities of information and retrieve them with lower latency than in traditional databases [Pok13]. For these reason, the deployment of these systems can involve from few to even thousands of machines depending on the amount of data to store and the SLA to achieve. By default Cassandra does not offer any autoscalable mechanism and, only recently, researchers have started to systematically investigate autoscaling methods for Cassandra [CM13, NSG+15, KDSS16, QCB16].

Autoscaling methods have been investigated for several years in cloud-native web applications [LBMAL14, QCB16]. Such methods help both cloud service providers and users to reduce operational costs and cope with workload variations [AEADE11]. Databases are also increasingly adapted to support autoscaling [CZ14, SWED16], but they can face long acquisition times for the new nodes due to the wait time of data synchronisation. Indeed, the time needed to add an entirely new node can take days if the dataset is very large, while at the same time affecting the performance of the existing nodes.

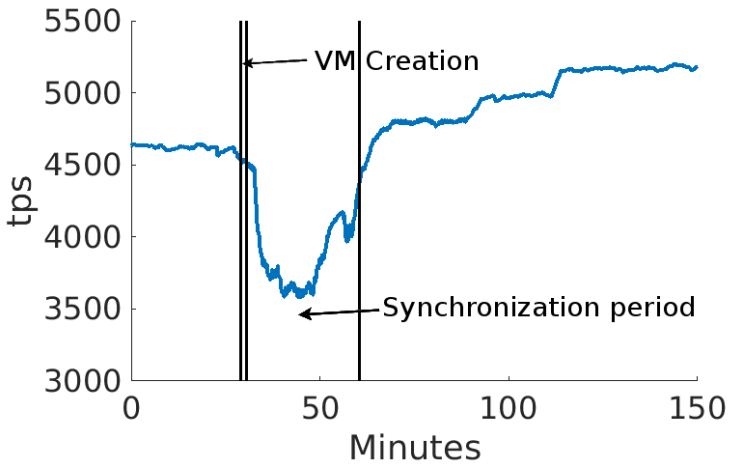



Figure 4.1: Data synchronisation period when a new node joins the cluster.

To illustrate the problem, we show in Figure 4.1 a measurement of the time required for Cassandra to add a new node to a cluster. In this example, the cluster runs on the Microsoft Azure cloud and consists of four virtual machines (VMs). Throughout the chapter, all experiments are run using the YCSB benchmark [CST+10]. Even though the cluster is small composed by 4 nodes with RF 2 and the data stored in the nodes is not too large, around 15GB per VM, the system spends almost 30 minutes to create and transfer the data to the newly instantiated VM. Only at the conclusion of this process the new node is finally acquired into the Cassandra cluster and becomes operational. On top of this, the system requires additional time to stabilise its performance with the new node. If multiple nodes are simultaneously added to the system, the situation further degrades, with the transfer time period becoming even longer. Clearly, this problem hinders the ability for an autoscaling system to quickly adapt Cassandra to the incoming workload.

In this chapter, we define a novel autoscaling method, called PAX, which relies on Cassandra's *hinted handoff* mechanism to efficiently add nodes to the cluster [Dat17] and we introduce proactive and reactive policies that control Cassandra using workload and data partitioning information. The hinted handoff mechanism exploited for autoscaling is the mechanism that Cassandra uses for synchronising pending writes to nodes that return online after some downtime. We argue that this mechanism can also be effectively used to enable autoscaling for Cassandra, provided that instead of creating and booting up new VMs one keeps idle a large

enough set of dormant (i.e., powered-down) VMs. Such VMs can be quickly synchronised to the cluster using the hinted handoff mechanism instead of deploying entirely new VMs.

After implementing autoscaling based on this mechanism, we show that PAX can effectively autoscale Cassandra as the rate of incoming queries varies over time. We introduce in particular a reactive and a proactive implementation of PAX, which scale resources based on CPU utilisation, workload demands and arrival rate forecasting. To further optimise autoscaling, we show that PAX can leverage query sampling to decide the best dormant VMs to activate, based on the data partitions they contain. Our experiments indicate that partition-aware node acquisition can provide substantial improvements in throughput up to 69% compared to any random cluster configuration that use the same number of active nodes.

We validate our approach through an extensive evaluation using time-varying arrival rates and different item popularity distributions. Among the main findings, we show that, compared to a system without autoscaling, PAX can deliver large cost savings without compromising the user QoS. In addition, thanks to the ARIMA predictive algorithm, the system seldom violates SLOs. To limit even more the SLOs violation, we present a more advanced autoscaling algorithm (OPAX) which considerably reduces the under-provisioning time and area.

The rest of the chapter is organised as follows. Section 4.2 reviews related work on elastic resource management for Cassandra. Section 4.3 introduces the Cassandra data partitioning and the hinted handoff mechanism. Section 4.4 presents our elastic architecture, the PAX and OPAX controllers. The tuning of autoscaling architecture and the experimental validation are given in Section 4.6 and 4.7. Section 4.9 the summary and conclusion.

## 4.2 Related work

Over the last few years, several NoSQL databases have been adapted to support elasticity resource management [KAB+11, DMVRT11, SWED16, KKR14, TKB+13, CMM+13, ASV13]. For instance, [TKB+13] presents a controller for elastic resource provisioning of HBase clusters using Markov Decision Processes (MDPs). Similarly, [CMM+13] defines a framework to auto-

matically reconfigure HBase nodes based on their access pattern. The work in [ASV13] shows instead a controller based on feed-forward and feedback signals for the Voldemort database.

Prior work on Cassandra autoscaling includes in particular [CM13], [NSG+15] and [KDSS16]. Both [CM13] and [NSG+15] present a reactive autoscaling mechanism. In particular, [CM13] develops a Cassandra controller that gathers node and workload performance data to calculate an exponentially-weighted moving average (EWMA) of the response time currently experienced by the users. When this moving average exceeds a pre-defined threshold, the controller adds a new node to the system. The work in [NSG+15] instead implements a controller driven using a Markov decision process (MDP) to model the cluster state and takes optimal autoscaling decisions. Compared to PAX, these works do not support proactive autoscaling and suffer high synchronisation latency due to the effect shown in Figure 4.1.

Recently, [KDSS16] presented a proactive controller for Cassandra that uses regression trees to predict latency. Upon exceeding a latency threshold, resources are scaled vertically to increase capacity and avoid SLO violations. Compared to this approach, PAX relies on CPU utilisation measurements, thus it is designed to optimise the infrastructure usage and cost, as opposed to user-perceived latency. Moreover, PAX adopts horizontal scaling and ARIMA time series forecasting of arrival rates. Recent work has shown that horizontal scaling tends to be more appropriate than vertical scaling for Cassandra databases [KKR14].

The cost of adding or removing nodes to Cassandra and other databases has been measured in [KKR14, KAB+11, DMVRT11, SWED16, RGVS+12, KAT+12]. For all the databases that do not use a shared file system, measurements indicate that the time required to add a new node is sensitive to the quantity of data stored in the entire database and the transmission rate between nodes. It is observed in [KKR14] that the usage of higher transmission rate affects the response time of the read operations.

## 4.3 Data recovery in Cassandra

In this section, we describe more in depth, first, the data partitioning mechanisms adopted by Apache Cassandra and then the data recovery mechanisms, in particular the hinted hand-off mechanism that are implemented on Cassandra.

### 4.3.1 Data partitioning

Cassandra's data are partitioned into smaller datasets that are stored locally to each node. Thus, contrary to other NoSQL databases such as HBase, Cassandra does not require a shared filesystem (e.g., HDFS) to exploit the data locality. A hash function is used to distribute the primary keys and associated data across the nodes. This is obtained partitioning the hash key range into sub-ranges called partitions (also called *TokenRanges*). In clusters without replication ($RF = 1$), each node $i$ can be configured to locally store $T_i$ unique partitions. The total number of unique partitions ($P$) is thus $P = \sum_{i=1}^{N} T_i$, where $N$ is the total number of nodes. For systems based on horizontal scaling with similar hardware resources, $T_i$ is set to the same value on all the nodes, since the VMs are usually identical. In this case, the $P$ calculation can be simplified as $P = T_i \cdot N$.

Using the replication ($RF > 1$), beside the unique partitions each node stores also some data replicas. The number of replicas that each node stored is defined by the $RF$ used. In this case, the total number of partitions available on a node $i$ is $P_i = T_i \cdot RF$, where RF is the replication factor. Then, we are able to calculate the total number of partitions like $P = RF \cdot \sum_{i=1}^{N} T_i$. For load balancing and high availability, the partitions are distributed randomly across the nodes with the only constraint that a given partition can be stored only once per server.

### 4.3.2 Hinted handoff mechanism

In this section, we present the mechanisms that Cassandra implemented to maintain the consistency across the data and the service that are activated when a node is not responding anymore

for any reason. Furthermore, we describe how we use these services to deploy our autoscaling system.

When a node is not responding for a defined period of time, the Cassandra cluster assumes that the node has failed. In this case, the hinted handoff mechanism will be tasked to facilitate the data recovering of the node. The mechanism works as follows. When the *request coordinator* believes that one of the cluster nodes has failed or it is overloaded of requests, pending writes are stored locally within a hinted handoff table. The request coordinator creates one table for each failed node and these tables are called *hints*. When the failed node becomes active again, all the nodes are notified of the cluster changes through the Cassandra gossip protocol and this trigger the second step of the hinted handoff mechanisms. In fact, before starting to serve client requests, the target receives from all the other active nodes the hint tables and applies the changes on its local copy of the data. When the transfer is completed, each node deletes the transferred hints tables restoring the normal Cassandra functionality.

The hinted handoff is a configurable mechanism. The database administrator can decide to activate or not the hint handoff, decide the amount of time for which each node needs to keep track of the data changes and the maximum *transfer rate* on which each node can send the hint tables. The tuning of this last parameter is the one which can impact more on the overall synchronisation time and performance of the cluster since it limits the amount of writes that the each node can send. Finding the right balance between the performance of the cluster while hints transfer is active and the time necessary to complete the task can be challenging. For our experiments, we use the default Cassandra configurations which expect to have the hinted handoff active and the transfer rate unlimited. However, regarding the recording period, for our experiments, we extended it to record the data up to twelve hours.

Figure 4.2 illustrates an example of hinted handoff in action. On this cluster composed by 8 nodes with replication factor 4, we temporary turn off 4 nodes. As shown in the figure, the size of the hint tables for the target node grows over time. For the purpose of this illustration, the red line represents the sum of the all hint tables for one target node recorded by all the other active nodes and the blue line the throughput of the overall system. After the first 30 minutes of

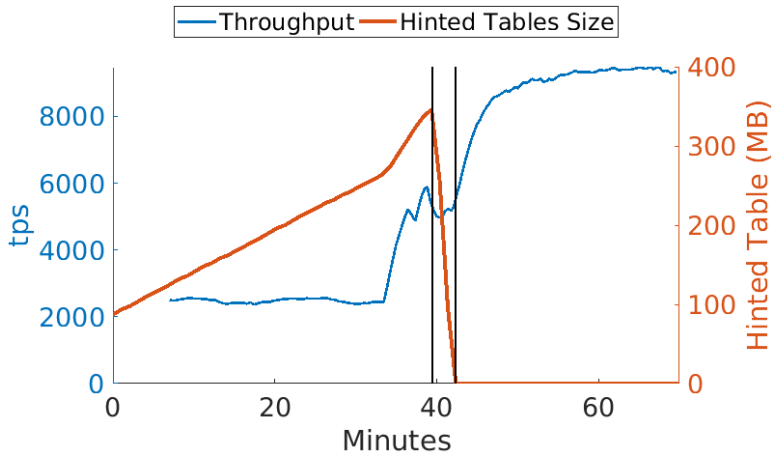



Figure 4.2: Data synchronisation through the hinted handoff.

the experiment we turn on all the stopped machines. When the target node becomes available again, each node immediately starts to send the hint tables to it and they are immediately applied by the node. The node spends only few minutes to apply the changes and to start to serve clients requests. In fact, the throughput of the system immediately starts to grow after the hinted handoff mechanisms is completed.

Beside hinted handoff Cassandra implements also another mechanism, called *write repair*, to repair the not up to date data. This mechanism is triggered by the request coordinator when, after the completion of a read requests, inconsistency between the returned values is detected. In this case, the request coordinator generates a new write requests with target the not up to date node to write the value associated to that specific record. This mechanism is a lighter process compared to hinted handoff and it is activated on demand.

Both hint handoff and write repair mechanisms are working at the same time in Cassandra to ensure data consistency across the nodes. For the purpose of autoscaling, we rely more on the hinted handoff because it is able to record the unperformed queries when some machines are not reachable and recover the data consistency across the cluster as soon as the node is running again. On the other hand, the write repair mechanism continues to work as normal verifying that the answered data to the clients are the most recent one and updating values if some nodes are not up to date.

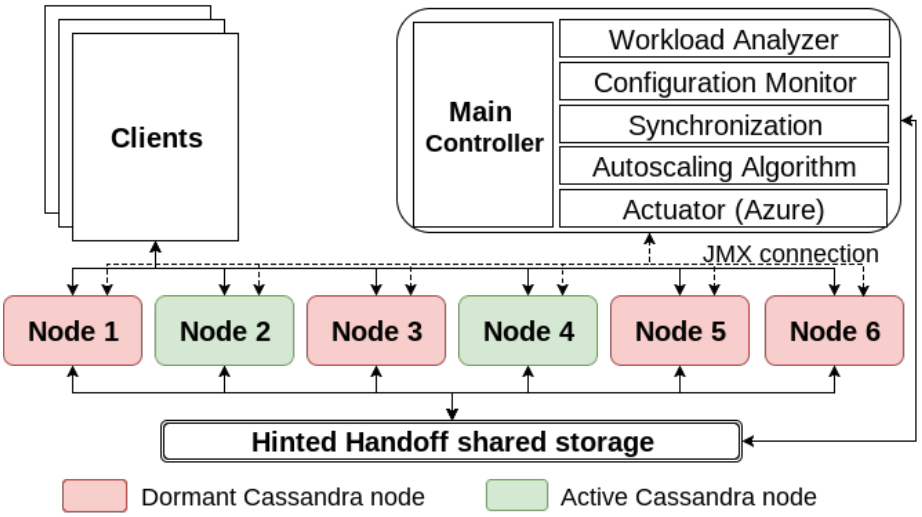



Figure 4.3: PAX architecture

## 4.4 PAX: Partition-aware autoscaling

In this section, we present PAX, the proposed partition-aware autoscaling method. Figure 4.3 illustrates PAX architectural setup, which relies on three components: i) the Cassandra cluster, consisting of a fixed set of nodes (VMs), which can be either in active or dormant (i.e., powered-down) state; ii) the controller, which analyses the workload measurements and actuates the autoscaling decisions; iii) a hinted handoff storage area, which archives the hints to be committed to the dormant nodes upon their return to the active state. In the following sections, we describe separately the controller component and how it interacts with its sub-components.

### 4.4.1 Controller

The core architectural element of PAX is the elasticity controller. The main aim of the controller is to ensure that the *average node utilisation* $U$ remains within a pre-defined CPU range $[U^-, U^+]$ at all times. We design the controller to be able to work without relying on any information about the cluster state. In fact, it gathers all the information directly from one of the active machines of the cluster. When the controller starts, it looks for an active Cassandra node using a provided list of IPs and, through JMX, it gathers the state of the cluster. Consequently, the controller is able to connect to each node and record the current CPU utilisation

and partitions managed by each node. All the other information are calculated at runtime by the controller.

Since PAX relies on horizontal scaling, it can be assumed that VMs have homogeneous sizes, and thus averaging utilisation across nodes is a well-defined metric. We also evaluate the possibility to use other target metrics with the PAX controller, such as the maximum utilisation across the nodes. However, we have experimentally observed that the maximum utilisation metric can lead to make the autoscaling more aggressive than the average node utilisation. This is due to the distribution of partitions across the nodes. In practice, we found that $U$ is a sufficient metric to implement effective autoscaling, as shown later in the experimental results. For this reason, we focus the implementation of PAX using the average node utilisation metric $U$ only.

All the *cluster configurations* generated by the controller need to be assessed before applying them to the environment. To be *valid*, a configuration needs to guarantee that all data partitions have at least $\mathrm{CL}^{max}$ replicas stored in the active nodes, where $\mathrm{CL}^{max}$ is maximum consistency level allowed for a query decided by the database administrator. PAX can operate either as a reactive controller or as a proactive one. In the reactive mode, when the system performance is out of target CPU utilization range, PAX interacts with the cloud provider API to adjust the configuration by starting and stopping VMs until $U$ returns within the range. The only exceptions is when the cluster cannot scale up or down any longer, either due to shortage of dormant nodes or because it has reached a configuration that cannot use less nodes without becoming invalid. On the other hand, using the proactive mode, PAX couples this control mechanism with a workload forecasting method based on ARIMA processes. To better illustrate the operation of the controller, in the next subsection we review the workload analysis and forecasting features that are supplied to the controller by the architecture.

### 4.4.2 Workload analyser (WA)

The workload analyser (WA) is responsible for monitoring the system, analysing the resource consumption data and the demand estimation. When PAX is configured as a proactive controller, WA is also responsible for workload forecasting.

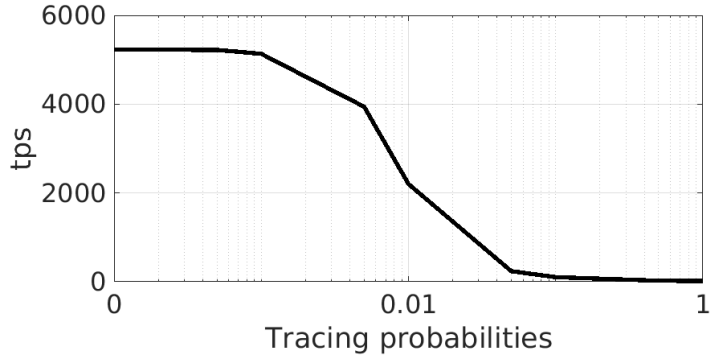



Figure 4.4: Overhead of the *tracing* tool.

WA monitors the type of requests issued to the cluster (read or write operations) and the requested primary keys. The main goal is to identify the partitions that are more frequently requested (*hot partitions*). This information is used by the PAX autoscaling algorithm to select the best dormant node to activate or, during scale down, to choose the active node to set as dormant. While PAX does not require initial training about the hot data distribution across the nodes, it can converge faster to a good configuration if supplied with an initial estimate of the hot partitions, based on historical data. However, the controller works correctly also without any prior information.

At runtime, the WA can obtain the list of hot partitions either using the *nodetool* or the *tracing* utilities shipped with Cassandra. The command *nodetool toppartitions* samples the activity of a Cassandra cluster for a specified period of time, returning the hot partitions. However, the command can degrade the database performance and, in one of the latest Cassandra versions we used (3.0.9), the tool frequently crashed. For these two reasons, we have used, in our implementation, the *tracing* tool.

The *tracing* tool is a troubleshooting utility to profile the internal operations that are executed by Cassandra to complete a query. In order to prevent performance degradation, the tool allows to randomly sample queries with a given probability. Figure 4.4 shows the tracing overhead we observe on the same testbed used in Figure 4.1 for different sampling probability values. If the sampling probability is chosen small enough, the system performance is not significantly affected by the background execution of the *tracing* tool. To determine the tracing probability value to use on the system, two different approaches can be applied. The first approach consists

in settings a static value, independent of the current workload of the system. On the other hand, the second approach can define some policies to dynamically adjust this value based on some performance metrics. For example, the user can define a range of possible values for the tracing tool and the workload analyser can, at run-time, dynamically change it based on the number of requests per second or the requests response time that the system is receiving from the clients. The higher is the number of incoming requests (or the response time) the lower will be the sampling rate. Using a range of values ensures that the system is not going to use a high sampling rate that can significantly compromise system performance. For our experiments, we choose to use the first approach and set the sampling probability for PAX to be 0.001, for which the system throughput is degraded by 2.9% on average. However, PAX can also work smoothly with lower sampling probabilities, although it can take a longer time to converge to an optimal configuration.

The WA is scheduled to be activated every minute. When starts, the WA reads and removes from the database all the tracing information provided by the *tracing* tool. For each query, the script extracts the primary key and calculates the partition on which the data resides. All the partitions are written into a file that is subsequently read to determine the list of hot partitions. Entries of this file that do not belong to the desired windows defined by the database administrator are deleted or moved to a separate file to be used as initial information for the WA.

### 4.4.3 Workload forecasting

The WA component also retrieves performance metrics for each node and predicts changes to the arrival rates of queries. In our implementation, performance metrics such as CPU utilisation and the total number of operations executed are retrieved from each active Cassandra node using JMX. However, other monitoring systems may be used.

To predict the future workload, WA maintains an AutoRegressive Integrated Moving Average (ARIMA) model. ARIMA is a method for non-stationary time series prediction that combines

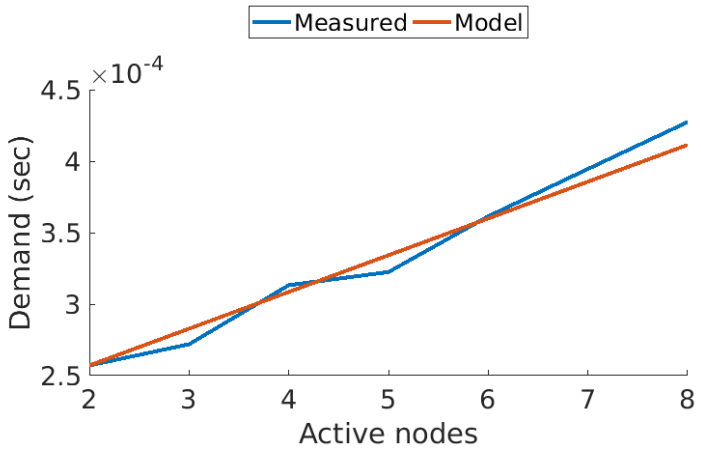



Figure 4.5: Mean service demand change with the number of active nodes.

an autoregressive and a moving average model. ARIMA is commonly applied on several autoscaling work [CMRB15, THIC11, BCH+14, FLWC12, ZZBH13]. In addition, compared to other time series forecasting models, it is easy to implement and parametrize. Anyway PAX is able to use any type of time series forecasting model that can provide an estimation of the near future workload that the cluster is going to receive based on the history of the database. We set the ARIMA forecasting time to consider the mean time required to boot up a dormant VM. In our experiments on Microsoft Azure based on VMs of class *A2*, we have measured this time to be around 3 minutes on average. ARIMA predictions are thus set at 3 minutes in the future.

WA predicts the global arrival rate $\lambda$ to the cluster, measured in transactions-per-second (tps). Since we are concerned with the mean utilisation across identical nodes, the value yields the predicted CPU utilisation as [LZGS84]

$$U^{pred} = \lambda \cdot D$$

where $D$ is the mean service demand of a node and it is estimated by linear regression of the current measurements of $U$ and $\lambda$. We have observed that the demand $D$ changes almost linearly with the number of active nodes, as shown in Figure 4.5. This is potentially due to the extra calculation and communication that each node needs to perform to be aware of cluster changes and to the decision that needs to be taken each time that the request coordinator needs

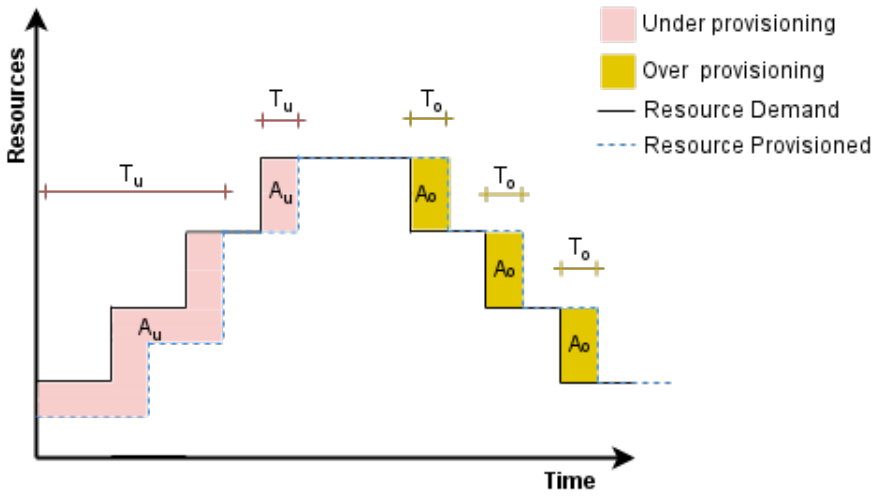



Figure 4.6: Over and under provisioning representation.

to execute a read request. In our tests, each additional active node increases $D$ on average by $\alpha = 10\%$. This correction is included upon forecasting the utilisation after an autoscaling decision. In general, the value of $\alpha$ is sensitive to the consistency level CL of the queries. In other setups it may requires runtime estimation by fitting to a line the obtained measurements of $U$ and $\lambda$ as the cluster configuration changes.

## 4.5 Autoscaling algorithms

Central to the PAX autoscaling algorithm are the decisions on: i) which nodes to scale; ii) how many nodes to scale; iii) when to trigger the autoscaling action. We discuss these aspects separately in the next subsections after the metrics definition used in the chapter.

### 4.5.1 Metrics

The metrics we collect are similar to those presented in [HKR13]. Figure 4.6 reports an example of under provisioning (in red) and over provisioning (in yellow) of the system. For either case, we calculate the time and the area on which the resource provisioned do not match the resources provided. In our case, we define the under-provisioning time $T_U$ as the time that the system spends above $U^+$ and the over-provisioning time $T_O$ as the time that the system spends below

$U^-$. $U^+$ and $U^-$ represent, respectively, the upper and lower bounds of the CPU range decided by the database administrator into which the system is considered in a stable state. We condition this to the system not using the maximum number of nodes, since we regard this situation as an error in static provisioning of the cluster size, rather than a shortcoming of the autoscaler. In addition, we also consider the under-provisioning area $A_U$ between the cluster utilisation and the upper bound of the target CPU range. This is computed only in periods when the utilisation is above $U^+$. Similar process is also used to calculate the over-provisioning area ($A_O$). The area is a quite important parameter on our analysis because it allows us to define the gap between the demanded and provisioned resources.

### 4.5.2 Data-aware node acquisition

When the PAX controller decides to take an action, so to increase or decrease the number of active nodes, it is necessary to identify the set of nodes that are going to be involved in the action. This is decided based on the information generated by the WA component by prioritising the acquisition of nodes including the hot data partitions.

PAX associates to each Cassandra node $i$ a score ($V_i$). The algorithm receives from the WA components the primary keys ($K$) and the key access rate $\lambda_k$ (requests per second) for the queries randomly sampled in the last control period. To identify the location of the data accessed by the sampled query, the primary key is hashed and the partition ($p \in P$) that contains the data is identified. For each of the node that stores that partition, the associate node value $V_i$ is then incremented by $\lambda_k$. Thus, for the $i$-th node, the node value is calculated as $V_i = \sum_{p \in P_i} \sum_{k \in K_p} \lambda_k$, where $P_i$ is the set of local partitions on the node $i$ and $K_p$ is the set of primary keys contained in the partition $p$. The values $V_i$ are used to decide the order in which dormant VMs are activated during scale-up or scale-down.

To assess the effectiveness of this data-aware scale-up approach, referred to as PAX (or data-aware *best*) selection, we show in Figure 4.7 a comparison against a method that selects the node with the *worst* $V_i$ score and with a method that picks the node at *random*. The experiment is conducted on a 16 nodes Cassandra cluster where the minimum number of active nodes is

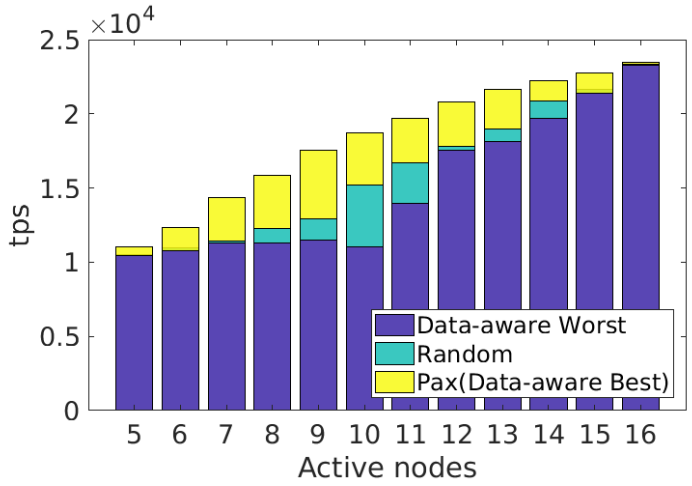
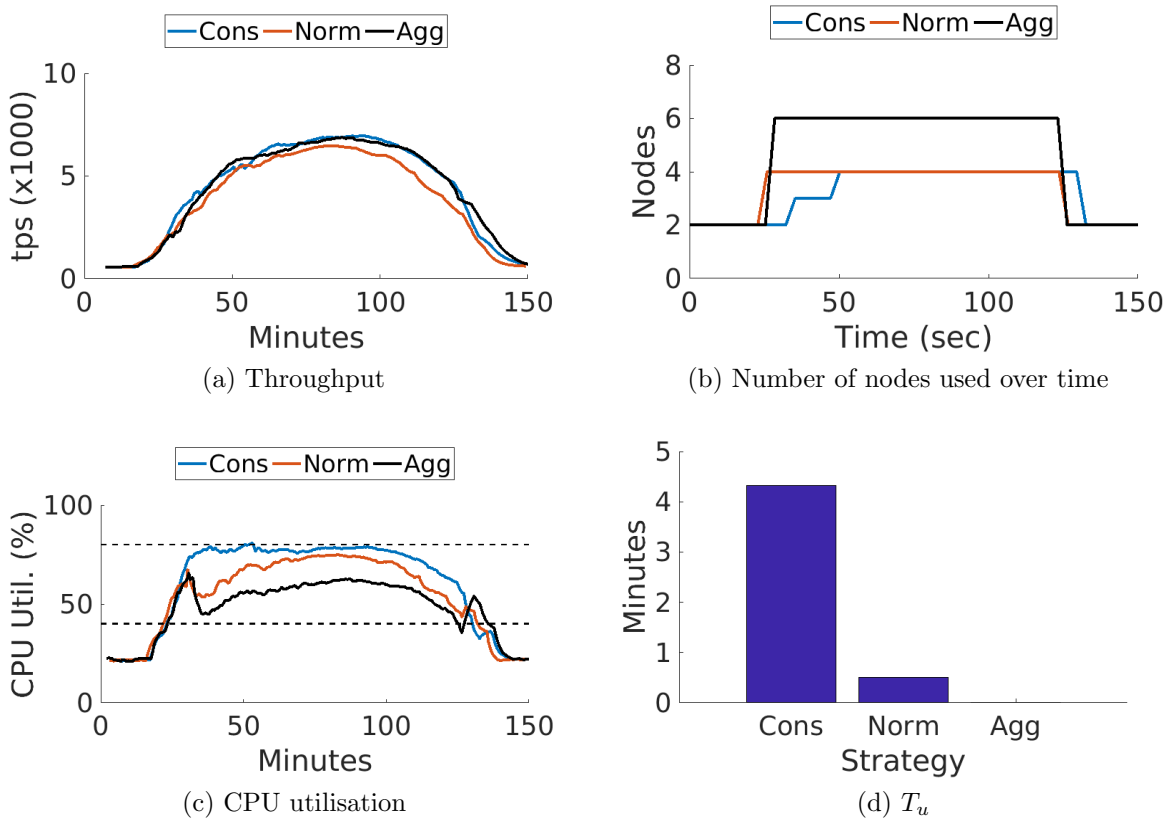



Figure 4.7: Gains due to data-aware node acquisition.

5 and the workload used a Zipfian popularity distribution for the data. To understand the best achievable performance, we use a static workload characterisation. The remaining nodes are activated one by one until all the nodes are active. Figure 4.7 reports the difference in throughput with different number of active nodes and using the different algorithms to select them.

The performance gap between the two algorithms is quite evident and reaches a maximum with 10 active nodes. We attribute the fact that the gap is maximal around the middle of the figure since the number of active hot partitions is significantly different. In fact, observing the data worst approach, during the first six step it does not increase the throughput because only nodes containing cold partitions has been activated. On the other hand, the PAX approach during this first six steps almost duplicates its throughput reaching a throughput of 19070 tps, while the worst selection has only 11220 tps. Progressively, the gap is reduced since, using the data worst, all the remaining dormant nodes contains hot partitions that can contribute to increase the performance of the system. The random algorithm shows as expected a performance in-between the two other methods, but it still fairly worse than the best one. Overall, this experiment confirms that data-awareness can produce visible gains during Cassandra autoscaling. We present in the next sections more advanced scenarios with dynamic workload characterisation and varying experimental setups.




(a) Throughput

(b) Number of nodes used over time

(c) CPU utilisation

(d) $T_u$

Figure 4.8: Aggressive strategies comparison.

### 4.5.3 Number of nodes to scale

PAX offers three strategies to control the number of nodes involved in a scaling action. These strategies are called: conservative, average and aggressive. Deciding the correct strategy for the selected environment is important since it can reduce the number of operations that the controller needs to perform and, more importantly, reduces the SLO violation.

In the case of a scale up with the conservative strategy, the smallest possible number of nodes that PAX predicts to bring back $U$ within the target range is used. On the other hand, the aggressive strategy activates the largest possible number of nodes such that the predicted $U$ remains between $U^-$ and $U^+$. The average strategy targets instead the center of the target range. Vice versa, during scale down, the conservative policy switches off the maximum number of nodes, while the aggressive ones powers down the minimum number of nodes.

Figure 4.8 presents experimental results for the three strategies. Figure 4.8(a) shows the

throughput changes over time. The conservative reacts slowly to these changing, performing many more scaling up actions than the other policies, as seen in 4.8(b). As visible from Figure 4.8(c), this means that the conservative strategy remains close to the upper bound of the utilisation band. Conversely, the aggressive strategy performs only 2 actions. However, it activates more nodes than the other strategies but it never violates the CPU upper bound, as shown in Figure 4.8(d). For this reason, we have decided to adopt in PAX by default the aggressive strategy.

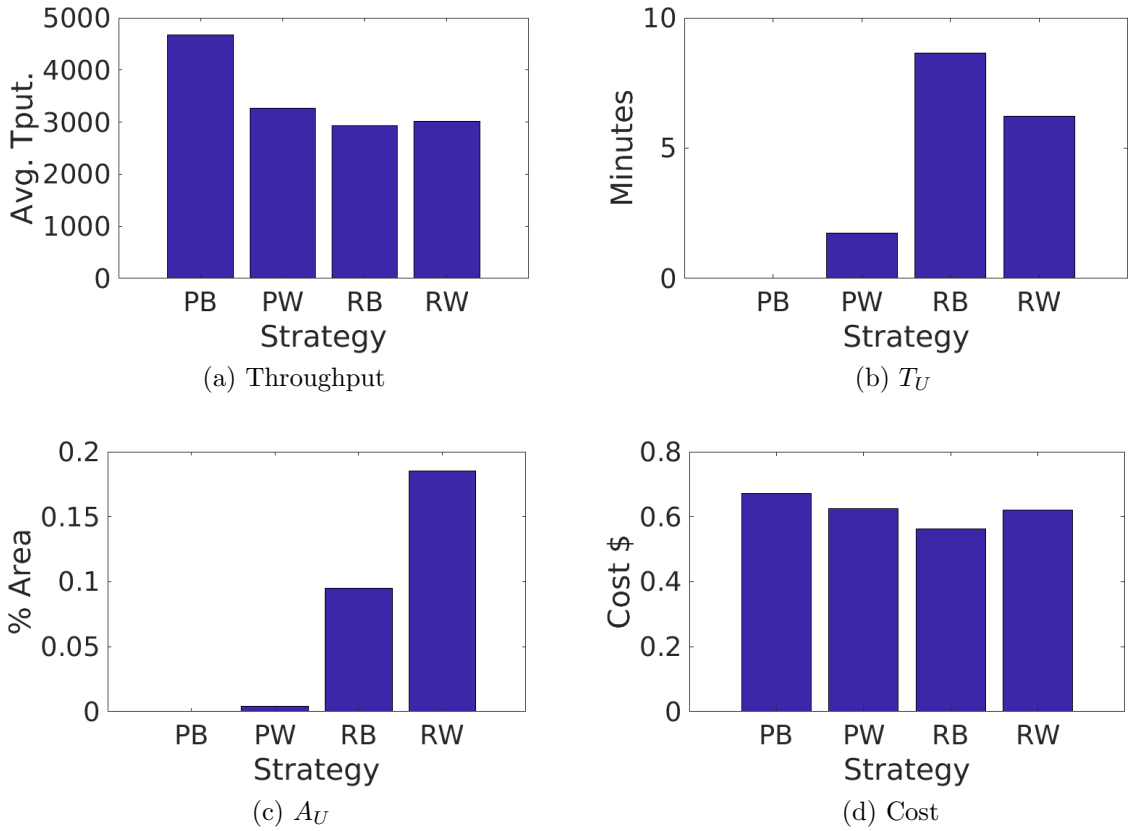
### 4.5.4 Triggering a scaling action

The data-aware acquisition and the strategies to select the number of nodes allow PAX to implement a scaling decision. This involves CPU utilisation prediction for all the possible node configurations, retaining only the valid ones within the target range. Among these, PAX selects the one with the desired number of nodes, based on the strategy selected, and with the best nodes, based on data-awareness.

As mentioned, PAX implements the decision in either reactive or proactive approach. In both cases, PAX requires the CPU trend to violate for at least 3 consecutive times, spaced by 1-minute intervals, the CPU bounds before the controller takes any action. This avoids unnecessary or inaccurate actions triggered by errors in the predictive model. Moreover, we set a stabilisation period of time of 10 minutes in-between any actions cannot be taken. This reduces the likelihood of taking incorrect actions during instabilities or fluctuations of the system measurements due to the recent changes.

Figure 4.9 compares the proactive and reactive implementations of PAX and also illustrates the impact of data-awareness on these by considering the best data-aware node acquisition of PAX against the worst-case acquisition method. The experiments run on a 8 nodes cluster with a 2 hours workload peaking at a maximum of 80 clients.

The results show that the reactive controller introduces some under provisioning time due to the lack of utilisation prediction. On the other hand, the proactive controller exhibits




(a) Throughput (b) $T_U$ (c) $A_U$ (d) Cost

Figure 4.9: Comparison between Proactive PAX (PB), Proactive Data-Aware Worst (PW), Reactive PAX (RB), Reactive Data-Aware Worst (RW).

negligible under-provisioning time and area. However, the proactive system anticipates some actions consuming more resources and with a higher experimental cost. The throughput results convincingly argue that the proactive data-aware controller performs much better than all the other methods.

Based on this analysis, we consider the reactive controller better suited in situations where the user wants to reduce the operational cost, when he is renting the VM. On the other hand, the proactive controller limits considerably the under-provisioning time of the system making it more appropriate in those situations where the user wants to reduce CPU utilisation, as in the case of providers exposing Cassandra services from within their own infrastructure. Given its increased ability to control the system, in the next sections, we will focus on evaluating the proactive version of PAX.

Table 4.1: Performance comparison results.

| | $T_i = 2$ | $T_i = 256$ |
|---|---|---|
| $N = 8$ | 8683 ops/sec | 9067 ops/sec |
| $N = 16$ | 10663 ops/sec | 10684 ops/sec |

Table 4.2: Minimum configuration size $M$ under different $T_i$ values. Experiments executed with a Cassandra cluster with $N = 8$ nodes, assuming a $CL^{max}$ of ONE

| | | $T_i$ | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 4 | 8 | 32 | 128 | 256 |
| RF=2 | $P_i$ | 2 | 4 | 8 | 16 | 64 | 256 | 512 |
| | $M$ | 4 | 4 | 5 | 6 | 7 | 7 | 7 |
| RF=4 | $P_i$ | 4 | 8 | 16 | 32 | 128 | 512 | 1024 |
| | $M$ | 2 | 2 | 3 | 3 | 4 | 5 | 5 |

## 4.6 Tuning the PAX Architecture

In this section, we provide additional details concerning the PAX architecture, such as the number of nodes to be used in the cluster and the setup of the hinted handoff storage mechanism. Upon instantiating the cluster, the administrator should decide several parameters such as the maximum number of nodes $N$, the RF, the maximum CL allowed for a query ($\text{CL}^{max}$), and the data partitioning setup. These decisions affect the flexibility of the autoscaling mechanism, i.e., the minimum number of machines $M$ that need to remain online at all times to ensure that the configuration remains valid. The parameters to consider are:

- **Total number of nodes**: The maximum number of nodes $N$ should be such that the cluster can achieve at least the maximum desired throughput when all nodes are online. Benchmarking may be used to estimate the $N$ parameter experimentally against a reference workload. $N$ can also be changed at runtime where needed, but as observed in Figure 4.1 the synchronisation time will be much longer than with the hinted-handoff mechanism, which requires only a few minutes.

- **Replication factor**: Normally, RF is chosen in production environments to be between 2 and 4. RF of 1 is usually not advised since, in case of failure of any node, the system may not be able to recover the data. Besides, RF 1 does not allow to have any kind of autoscaling system since it requires that all the machines are active at any time.

Probabilistic methods have been devised to analyse the influence of RF on the system availability, resilience to malicious users and identify an appropriate assignment [YG07, LGR15, YGN06].

- **Data partitioning**: The RF and the number of partitions per node $T_i$ are static properties of the cluster decided upon its creation and the initial loading of the dataset. They determine the set of valid configurations for a cluster and thus the flexibility of the autoscaling mechanism. Although $T_i$ does not significantly affect performance, as shown in Table 4.1 for a 8-node cluster, it influences the placement of data on the nodes, and thus how many nodes can be turned off by the autoscaling controller while remaining in a valid configuration.

Let's consider, for example, the case where the replication factor and the number of nodes are the same and the database administrator decided to use $CL^{max}$ ONE. In this case, the autoscaling system can reach its maximum flexibility since it can activate from one to $N$ machines due to the fact that all the data are stored on all the nodes and the maximum consistency level used is ONE. If we change the $CL^{max}$, the number of minimum number of active machines will be the value of $CL^{max}$. On the other hand, this cluster requires the provisioning of an higher storage space since all the machines need to store all the data of the database. A completed opposite situation is represented when the RF is ONE and the only valid configuration is the one with all the machines active since the data is the only copy on the cluster.

In all the other possible configurations of the cluster, the data partitioning also come in place and determine the flexibility of the system as well as the replication factor and maximum consistency level because the distribution and location of the replicas inside the cluster determine the system flexibility. For example consider now a cluster that with N nodes and $RF = 2$ that uses $CL^{max}$ ONE. The smallest valid possible configuration could be represented by $N/RF$, so in this case half of the cluster, but in reality this is also related to the location of this copy. If $T_i$ is sufficiently large or uses the default Cassandra vnodes value (256), the system requires almost all the nodes to be active losing the autoscaling flexibility. For this reason, finding the

correct balance between the data partitioning and replication factors is essential to activate a good autoscaling flexibility. However, the only way to find the real minimum valid configuration is through experiments or simulation.

The dependence between these two factors is illustrated in Table 4.2. Here, we run a set of 14 experiments, changing the $T_i$ and RF values. For example, the first column considers the case where each node contains $T_i = 1$ unique partitions. In this case, out of the $N = 8$ nodes, 4 nodes should always remain online if $RF = 2$, but this reduces to 2 if $RF = 4$. However, larger RF values increase costs, since more storage capacity and nodes will be needed to replicate the data.

In addition, an optional variable that the user may be interested in considering is the availability of the machines provided by the cloud provider. Usually inside the desired *region* chosen by the user, the cloud provider offers the opportunity to use different *zones* to enhance the availability and reliability of the infrastructure. Placing the application or database on different zones can ensure that, even if a zone is unreachable for some time, the service will still be delivered as normal. If the user is interested in applying this concept to PAX as well, after he has looked at all the possible valid configurations obtained with only the previous variables, he can exclude all the configurations that do not satisfy the high availability requirements of the cluster. Indeed, this parameter can limit considerably the flexibility of the system. So, since we do not have the possibility to use a cluster larger enough to test this scenario, we decided to not include this parameter inside our experimentation.

### 4.6.1 Hinted handoff storage

To ensure that each node can retrieve all hinted handoff tables anytime, we recommend to setup a shared storage area among the cluster nodes. For this storage solution, we decided to use the shared file storage system provided by our cloud provider to leverage on the high availability of their system and redundancy of the data across zone and regions. By default, the hints tables are stored locally on each Cassandra node disk. However, it may happen that some tables are

stored on nodes that become dormant, causing the risk that a newly activated node does not find all hinted handoff tables required for its synchronisation.

A shared storage area avoids the above issue. Within this area, each node can store its tables in a specific folder, so that they are always available for retrieval during scale up operations. If one of the nodes changes status to dormant, the hint files are then moved each one on the specific folder of the target node. So, when a dormant node is powered up, it receives the data from all the active nodes and then it applies also the tables that it finds on its shared folder.

Although we did not experience similar cases, it is conceivable that if some nodes remain dormant long enough, the size of the hinted handoff tables may grow large enough to become an issue for both performance and cost. Several strategies are possible to mitigate this risk. One possible solution could be to adopt a hybrid architecture, where most hints are stored locally to the nodes and only the ones of the dormant nodes are in the storage area. A simpler alternative consists in periodically activating the dormant nodes to allow them to synchronise the pending hints. In addition, this option can decrease the required time necessary to boot up the machine when it is really necessary. Furthermore, platforms such as Azure allows you to pay only for the minutes effectively used, so cost should be contained.

A possible limitation, which we have not experienced during our experiments, is the bandwidth bottleneck that this storage system can have in place. If the PAX deployment is large enough, several nodes can have the necessity to write or read these files on the common shared storage. We believe that the limits set by Microsoft Azure for the standard file storage service are reasonably high for normal Cassandra size deployments. However, if the user incurs in such issue, usually cloud providers offer also a premium version of them with the possibility to reserve the number of IOPS that the cluster needs, guaranteeing that the nodes can quickly access the data without incurring in the additional delays during the booting up process which impacts on the system costs and performance. The usage of other distributed storage systems has been also investigated but, from the best of our knowledge, these systems will impact considerably more on the cost of the entire infrastructure.

Table 4.3: Testbed configuration used for the controller evaluation.

| | Node | YCSB client | Controller |
|---|---|---|---|
| Number of VMs | 8 | 4 | 1 |
| VM type | General purpose | | |
| O.S. | Ubuntu 16.04.2 LTS | | |
| vCPUs | 2 | 2 | 4 |
| Memory | 3.5GB | 3.5GB | 7GB |
| O.S. Disk | 30GB Read/Write Host Caching | | |
| Data Disk | 80GB (No Caching) | none | none |

Table 4.4: YCSB Workload characteristic used for the system evaluation

| Workload | Read | Write | Distribution |
|---|---|---|---|
| A | 50% | 50% | Zipfian |
| B | 95% | 5% | Zipfian |
| C | 100% | 0% | Zipfian |
| G | 100% | 0% | Latest |

## 4.7 Performance Evaluation

### 4.7.1 Methodology

The evaluation of our elastic system is conducted on Microsoft Azure cloud with a Cassandra cluster composed of 8 nodes. The hardware characteristics of the virtual machines are presented in Table 4.3. On each Cassandra node is installed Sun Java 1.8 and Cassandra 3.0.9. In the original Cassandra configuration, we set in the configuration file the *num_tokens* to 2 to allow us to have a larger elastic range (from 2 to 8 VMs) with a CL of ONE and the hints folder is redirected to the shared disk managed by Azure. We also install and enable Jolokia to expose the JMX metrics over HTTP. The Cassandra database is loaded with a total of 180 GBs of data using a replication factor of 4.

Compared to the state-of-the-art in Cassandra autoscaling presented in Section 4.2, our setup has a larger database with a higher replication factor. Using this architecture, these factors significantly impact on the data synchronisation time necessary to the system to change the configuration, since the amount of data that each node needs to transfer is bigger.

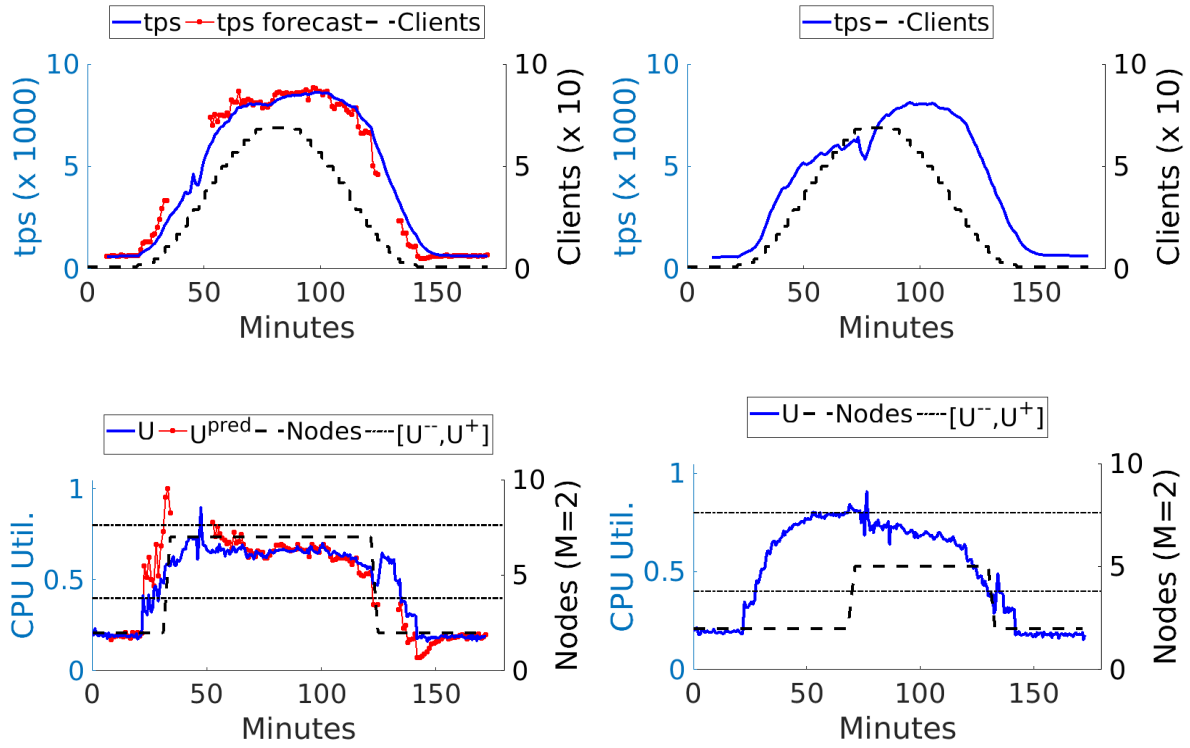
The workload generator machines run the YCSB benchmark [CST+10]. The YCSB workload

characteristics are summarised in Table 4.4. The defined workloads are characterised by different proportion of read and write operations. Most of the workloads are using the *Zipfian* distribution with the exception of the workload G which uses the *Latest* distribution. Zipfian and Latest distributions are very similar, but they select differently the target primary keys. Using the Latest distribution, the most recent keys becomes the most popular and they are queried more frequently. On the other hand, the popularity of the items in Zipfian is not influenced by new records. Workload generator is deployed on 4 virtual machines and one VM for the PAX controller. The PAX controller VM is independent and not able to compromise the Cassandra stability so, in case of failure, the controller can be quickly rebooted and resumes execution within a few minutes, during which the database operates as normal.

Each experiment is represented by two plots. The top figure shows the increase in arrival rate (in tps) and clients. The bottom figure shows the response of the PAX controller in terms of number of nodes, actual utilisation $U$ of the testbed and predicted utilisation values by the ARIMA forecasting. The predicted throughput and CPU are represented by the dot marker. In-between any two markers, *no* prediction is performed since the controller awaits that the system stabilises to retrain the ARIMA process and resume the forecasting.

### 4.7.2 Comparing proactive and reactive approaches

Figure 4.10a and Figure 4.10b compares PAX using the proactive and reactive controller. The experiment uses a workload B, lasting two hours and exhibiting a peak in the number of active clients. The figures show that both strategies are able to scale and satisfy the demands of resources. However, the reactive approach scales later in time compared to proactive and it also use less machines. In addition, by using less resources, the average CPU utilisation of the reactive method is higher than the proactive case. This has also some impact on the overall system throughput where the system satisfies the maximum load later compared to the proactive one. On the other hand, the reactive environment consumes less resources, saving around 0.126$ per hour. The proactive controller reduces significantly the under-provisioning of the system and it supports better the time-varying workload.




(a) proactive (b) reactive

Figure 4.10: PAX benchmark with a peak of maximum 80 clients using workload B.

### 4.7.3 Step response and overlapped peaks

Figure 4.11a presents the performance of the PAX controller under a step increase of 80 clients, using the workload A. Since the system does not have any information about the future, the controller is, as expected, not able to anticipate the sudden step increase but, immediately after, it reacts correctly to it by starting new VMs that avoid the utilisation to step out of the target range, except for a negligible period as reported in Table 4.5.

Differently, Figure 4.11b presents an experiment with two successive peaks in the number of clients and based on workload C. As the clients grow, the CPU utilisation sharply reaches the upper bound. As this is a rather slow growth pace, the ARIMA predictor suggests that this peak can be handled with the current testbed and this is indeed the case. However, as the second larger and growing faster peak arrives, the ARIMA controller is able to anticipate it in the initial stages. In fact, shortly after 100 minutes, PAX activates other 3 VMs to handle the peak workload effectively. Even if the system presents a higher $T_u$ time compared to the other proactive experiments (Table 4.5), the $A_u$ is very low meaning that the system is really

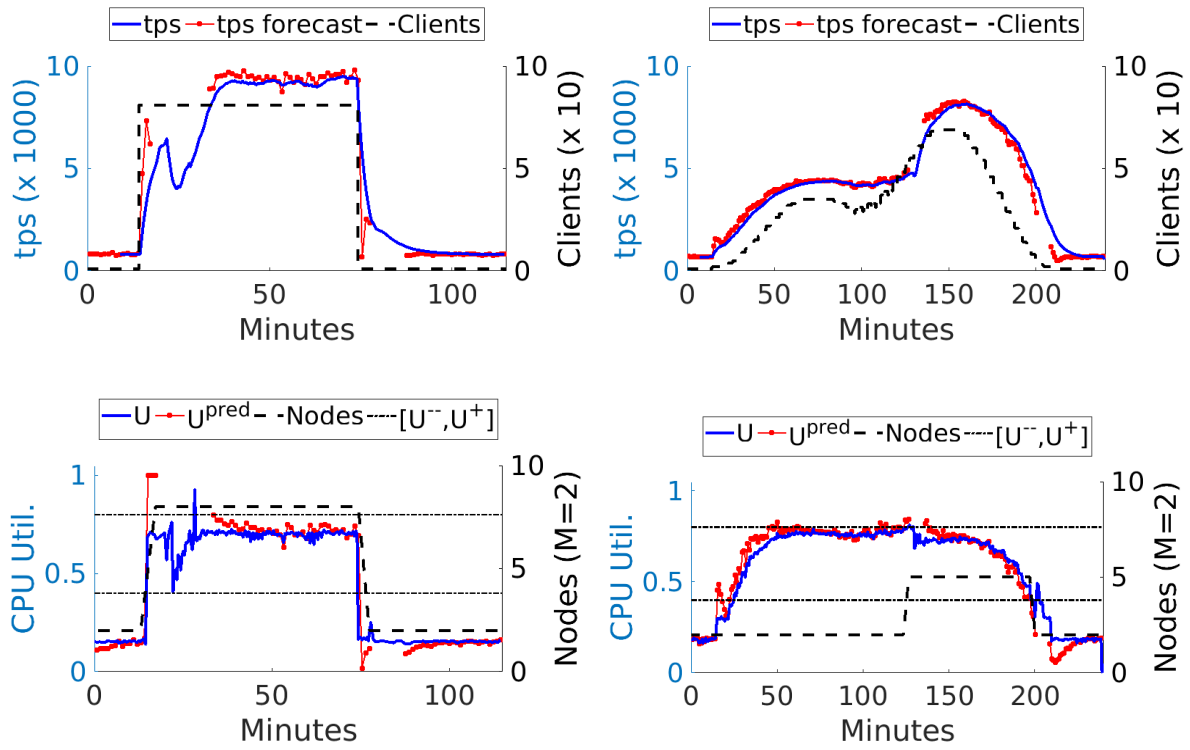



Figure 4.11: PAX controller response to a) a step of 80 clients starts issuing YCSB workload A; b) two overlapped peaks and workload C.

close to the $U^+$ bound. In addition, this experiment presents the higher cost saving of 61.42% compared to a traditional Cassandra implementation where all the nodes are always active.

### 4.7.4 Architecture change

We now show the ability of the WA query sampling to trigger actions in response to a change of the query mix issued by the clients. The results are presented in Figure 4.12a. During the experiment execution, the workload generated by YCSB changes from workload C to G, modifying the primary key access rates. These two workloads are composed both by read only operations but they use different distributions. Zipfian and Latest distributions are very similar but they select differently the target primary keys. Using the Latest distribution, the most recent keys become the more popular and they are queries more frequently. On the other hand, Zipfian popularity of the items it is not influenced by new records.

Few minutes after the start of the figure, the PAX controller, using the WA information,

Table 4.5: Evaluation results for PAX and OPAX.

| | max $N$ | $T_U$ | $A_U$ | $/min | $ saving |
|---|---|---|---|---|---|
| Fig. 4.10a | 7 | 30s (0.41%) | 0.07% | 0.0059 | 33.56% |
| Fig. 4.10b | 5 | 505s (3.54%) | 0.20% | 0.0038 | 56.36% |
| Fig. 4.11a | 8 | 21s (0.3%) | 0.12% | 0.0074 | 11.53% |
| Fig. 4.11b | 5 | 184s (1.28%) | 0.007% | 0.0037 | 61.42% |
| Fig. 4.13a | 8 | 0s (0.0%) | 0.0% | 0.0085 | 15.34% |
| Fig. 4.13b | 8 | 30s (0.416%) | 0.01% | 0.0077 | 17.00% |

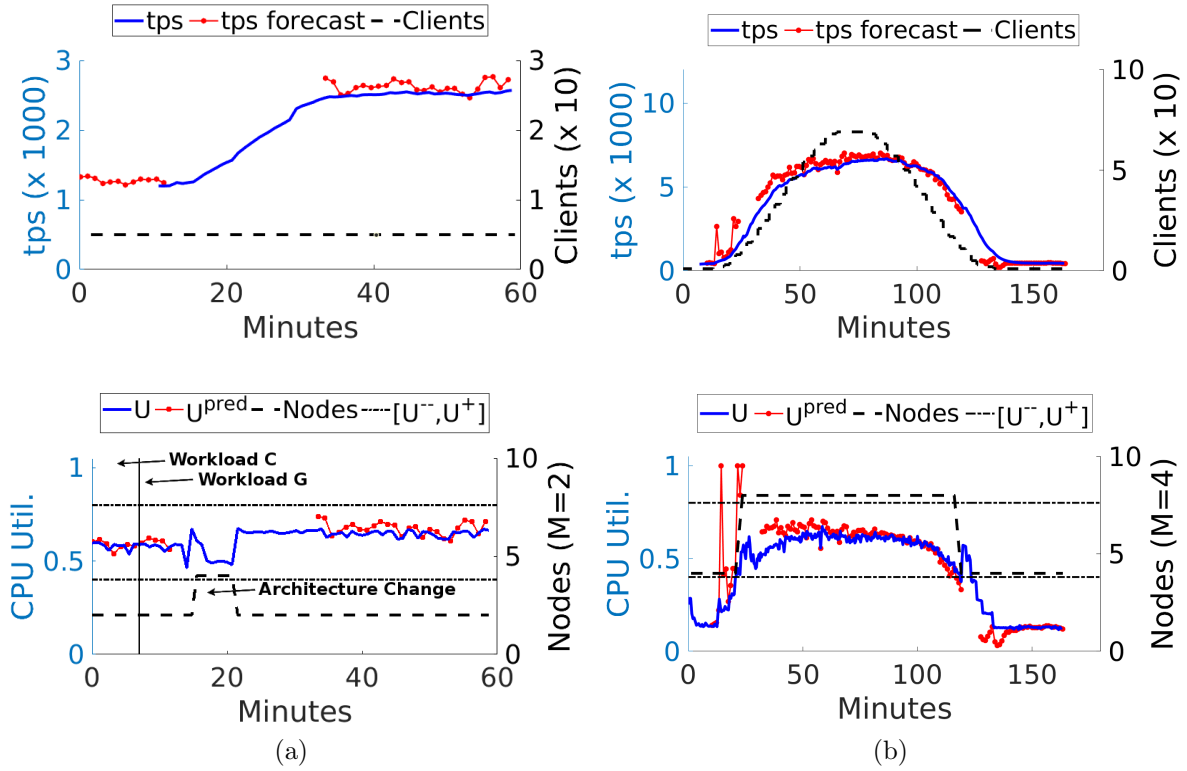



Figure 4.12: PAX controller response to a) changes in the hot partitions; b) a different consistency level (CL=TWO).

identifies a better configuration for the G workload. When a new configuration is detected, the PAX controller activates it and during the stabilisation period the two configurations are both active at the same time. Then the VMs of the old configuration become dormant. The configuration changes significantly benefits throughput, which increases from 1190 tps to 2500 tps. This illustrates the benefits of partition-aware autoscaling in Cassandra.

### 4.7.5 Different Consistency Level

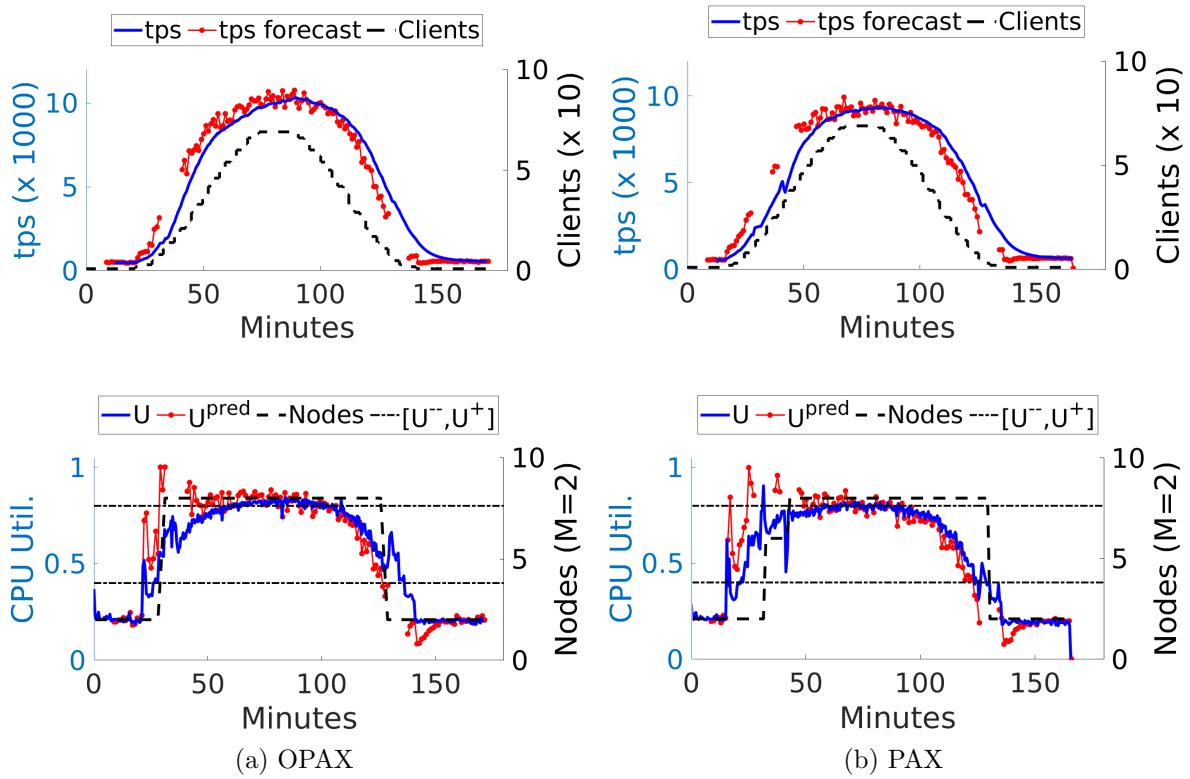
Figure 4.12b reports the case where a different consistency level is applied to the queries. In fact, instead of using the CL equal to ONE, we set the YCSB to perform only queries with consistency level TWO using the workload B, so two copies of the same data needs to be read or write to complete the request.

This experiment shows that PAX is able to work regardless of the consistency levels apply so, the database administrator is free to decide what kind of consistency level to apply to the queries. However, increasing the consistency level, also the number of minimum active nodes increases and the environment becomes more expensive. For the experiment presented in Figure 4.12b the minimum number of active nodes is set to four. Since Cassandra is able to find enough replicas of the data it is able to work properly and PAX is able to scale out when more clients are connected to the cluster.

## 4.8 OPAX strategy

We try to improve the PAX controller defining a more sophisticate strategy, called Holistic PAX (OPAX). Differently from PAX, OPAX takes in consideration all the factors at the same time to look for possible configurations that can potentially perform better than the one returned by PAX. In addition, OPAX is able to predict the CPU utilisation of all the active machines involved after the scaling action. OPAX relies on the same information used by PAX but it processes them in a different way.

When PAX autoscaling is triggered, the strategy starts to first understand the number of machines needed to support the workload and then, based on this number, it decides to which machines change the status. On the other hand, OPAX generates a list with all the possible configurations that can be deployed, which has a valid configuration for the system, and then for each configuration, it calculates the CPU utilisation for all the active nodes and averages them.




(a) OPAX (b) PAX

Figure 4.13: OPAX and PAX comparison with a peak of maximum 80 clients using workload A

The average CPU utilisation of a possible configuration is calculated gathering the information provided by the WA, calculating the number of requests that each partition is receiving and splitting them across the number of available partitions that the configuration has. Let us assume that $\lambda_p$ is the access rate to the partition $p$ and $p_a$ the number of active partitions across the entire cluster. We can then calculate the new access rate to the partition ($\hat{\lambda}_p$) with the selected configuration like $\hat{\lambda}_p = \frac{\lambda_p}{p_a}$. Using this information we are able to calculate the CPU utilisation of the node $i$ using the formula $U_i = \theta \cdot X_i$ where $\theta$ is the demand of the system and $X_i$ is the predicted throughput for the node $i$. The predicted throughput is calculated as $X_i = X \cdot \sum_{p \in P_i} \hat{\lambda}_p$ where the $X$ is the overall system throughput and $P_i$ represent the set of partitions stored locally on $i$-th node.

Once the evaluation of all the valid configurations is finished, OPAX considers only the configurations that return an average CPU utilisation value inside the user defined range. The final configuration that is selected to be actuated, is chosen by the aggressive strategy based on the user setting. For our experience, the normal strategy is the one which performs better with

OPAX.

Now we want to present a comparison between the PAX and OPAX algorithms. These experiments are conducted using a peak of maximum 80 clients and using the workload A. Figure 4.13a shows the OPAX autoscaling behaviour while PAX is shown in Figure 4.13b.

The two algorithms are performing very similarly. Both used all the available nodes to support the maximum load of the system. However, PAX reaches the 8 nodes doing an intermediate step of 6 nodes. This can potentially reduce the cost of the infrastructure but it can introduce some SLO violations as show in the figure. On the other hand, increasing directly to 8 nodes, OPAX never violates the SLO during this experiment.

A different aspect to consider between PAX and OPAX is the complexity of the OPAX algorithm. Since we used a small cluster, we have not noticed any major difference between the two. However, since OPAX has the necessity to generate and evaluate all the possible valid configurations, in the worst case scenario it needs to evaluate

$$\prod_{M}^{N} \frac{N!}{(N-R)!R!} \tag{4.1}$$

where $M$ is the minimum number of nodes of a valid configuration that, in the most optimistic case, it is represented by the $CL^{max}$ to ensure to be satisfied. However, some factors can influence the number of configurations to analyse, speeding up the process and reducing the time to identify the best configuration. For example, in the case that the cluster has $\frac{N}{2}$ active nodes and the load is increasing, so the system needs to scale up, OPAX can set $M$ to start from the number of active nodes or a value around that number. A similar process can be also defined when the system needs to scale down. In general, different policies can be defined but, due to the inability to test them, we decided to investigate them on some future work.

## 4.9 Summary and Conclusion

In this chapter, we have proposed two novel auto-scaling systems for Cassandra called PAX and OPAX. Both leverages the hinted-handoff mechanism of Cassandra to reduce the data synchronisation period when a new node is added to the cluster. Based on this, we have defined reactive and proactive controllers that profile the current workload and use this information to activate the Cassandra nodes that store hot data partitions. We have found that both reactive and proactive implementations are useful in practice. The reactive method uses less VMs but incurring more frequently under-provisioning, while the proactive allows negligible violations of the target utilisation. We evaluate PAX under different scenario and in all the situations it is able to address the demand of resources efficiently.

Furthermore, we develop a more sophisticated autoscaling system that reduces even more the SLO violation compared to PAX. Both the autoscaling systems are able to identify the correct architecture to deploy based on the current workload profiler information and to optimise the infrastructure costs. Based on our experiments, the proposed autoscaling system is able to reduce up to 60% the operational costs.

# Chapter 5

# SD: a Divergence-based Estimation Method for Service Demands

## 5.1 Introduction

Over the last decade, models to predict performance behaviour of cloud systems have been increasingly used to drive automated resource management [ACC+14]. In particular, the constantly growing trend of retrieving and storing monitoring data in production systems for analytics has also paved the way to a new generation of automated methods to parametrize performance models from empirical datasets [PHK17, SCBK15]. In traditional capacity planning methodologies, performance models are developed and validated on a test environment which may not reflect the hardware or workload characteristics of the real system in production requiring manual tuning and frequently leading to incorrect decisions at run-time [MZR+07, CMS10, MPGSL06]. By developing automated methods for demand estimation, one may instead parameterize performance and reliability models for a given system directly in the production environment, thus addressing the shortcomings of classic methods and concurrently supporting other system components, such as cloud auto-scaling controllers [DBC18, HHL12, DWS12]. A common problem arising in performance model estimation is the accurate inference of *service demands*, which are the average computational requirements placed by different types of

requests within a system. Existing estimation algorithms obtain demands from steady-state metrics such as throughput, response times[LXMZ03, UPS+07, PPSC13] , and resource utilization [NKJT09, BKK09, PSST08, KZ06, ZCS07]. However, existing techniques can have erratic performance and it is difficult to identify a method that is best in all cases [SCBK15]. The most commonly used demand estimation methods accept in input *mean* performance metrics, thus not fully exploiting during the inference process of a large part of the information available within a measured dataset. For example, although more challenging mathematically, methods that exploit observed response time distributions have been proposed to address this shortcoming [SCBK15]. However, the applicability of these techniques is often limited by their scalability, as they can require a numerical solution to the underpinning queueing models using rather computationally-intensive approaches such as absorbing Markov chains or fluid differential equations and non-linear optimisation [KPSCD09, PHK17].

In this chapter, we instead propose a demand estimation method that requires to collect *marginal state probabilities* for the number of requests processed by individual nodes of a cloud system. The marginal state probability defines the probability to observe a resource with a particular mixture of requests in execution. This is different from the joint probability which considers the state of the entire system. While probabilistic methods have appeared in recent work [WCKN18], our proposal differs as we wish to carry out a more advanced probabilistic analysis that takes into account the execution workflow of a request, in order to more accurately estimate demand at various stages of operation. We assume that each job is tagged with a class of service at a given time. Workflows are then described in terms of a sequence of *class-switching* steps for jobs that visit one or more resources in the system [BCMP75, BGDMT06] . That is, upon moving from a node to another, a job can switch its class so to represent a different phase of execution.

Class-switching has received a limited degree of attention in prior work on demand estimation, possibly with the exception of [PHK17] which considers it in the context of a single multi-server resource. However, workflows arise commonly in distributed systems composed by multiple resources, which is the case we consider in this work. For example, a request that visits the same resource multiple times, asking statistically different computational requirements at every visit

is naturally captured in terms of a job that switches its class in-between visits. For example, a job that visits multiple times the same node, requesting different execution requirements at each visit, may be modelled by assuming that the job switches class in-between visits. Distributed NoSQL databases such as Apache Cassandra provide an example, in which class-switching can be used to express a workflow of execution through multiple nodes in order to retrieve the data needed by a query for its completion [DCS17].

Despite their practical importance, class-switching models can be difficult to deal with due to state-space explosion, which is more rapid than in standard multiclass models. In order to increase their tractability, we consider class-switching in the context of product-form queueing network theory [BCMP75], which allows one to exactly transform a model with class-switching into a multiclass model without class-switching [Zah79, BGDMT06]. Despite this equivalence result, we notice that in the presence of load-dependence there appear to be no exact formulas available to perform this mapping for computing marginal state probabilities, a gap that we also overcome in this work.

Leveraging the proposed methods to obtain marginal probabilities in the presence of class-switching, we propose a new demand estimation approach, based on information-theoretic divergence measures, called *State Divergence* (SD) estimation. SD seeks to minimise the divergence between marginal state probabilities and their corresponding empirical estimates, in order to produce accurate estimates for the demands. Because marginals capture each job class, without the aggregation implicit in the transformations from class-switching models to multiclass models without class-switching, it can explicitly capture the actual mix of requests in execution at a resource more accurately than using the aggregated model. The rationale for this method is that, by minimising state divergence, one should obtain model parameters that accurately capture the stationary behaviour of the system, as reflected by occupancy in a fraction of its observable states, more accurately than by looking only at its output performance metrics, such as throughput, response times, or CPU utilisation.

We validate the SD estimation algorithm through several randomly generated models and with a case study conducted using the Apache Cassandra NoSQL database [LM10]. Our results

demonstrate that SD can significantly reduce errors in performance prediction compared to state-of-the-art algorithms which do not explicitly account for class-switching. The obtained demands are able to match the system state while reproducing a more realistic behaviour in the model compared to state-of-the-art estimation algorithms.

The rest of the chapter is organised as follows: in Section 5.2 we provide a motivating example. Section 5.3 presents a novel probabilistic formula we have developed to analyse systems with class-switching. The SD algorithm is then developed in Section 5.4, where we also present several possible divergence measures that can be used with our method. In Section 5.5 and Section 5.6, we give experimental results on Apache Cassandra and random model instances. Finally, Section 5.7, the conclusions are presented.

## 5.2 Motivation Example

Over the years, several inference algorithms have been developed to parameterize performance models of cloud systems [SCBK15]. For the purpose of this section, we focus our attention on the complete information (CI) algorithm [PPSC13]. This algorithm estimates the demand taking, as input, samples of the arrival rate and execution time of the system request. We consider this algorithm because it supports multi-threading, multi-class and class-switching.

To better understand and illustrate the limitation of an existing method such as CI when the class-switch is in use, we want to analyse the difference between the marginal state probabilities of a real system and the performance model parameterized using the service demands obtained by CI. For the purpose of this experiment, we use a Cassandra cluster composed of three nodes, deployed on Microsoft Azure. As workload generator, we run YCSB [CST+10] with 10 clients that continuously perform read queries to the database with a consistency level *ONE*. While the system is working, some network traffic is recorded from one node and then processed with the CI demand estimation algorithm. Then, the resulting demands are set into the Cassandra model presented in [DCS17] for recording the marginal state probability of the model of a single Cassandra node and compare it with the real node.

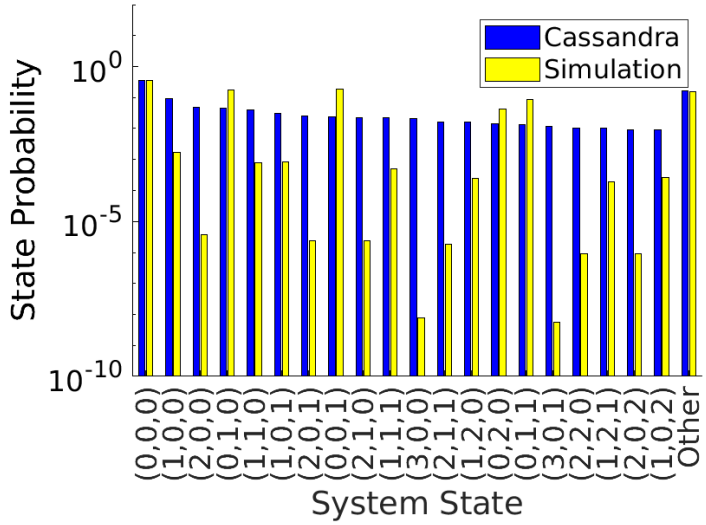



Figure 5.1: Marginal probability difference between Cassandra and Simulation.

Figure 5.1 compares the two marginal state probabilities. The horizontal axis shows the system state of a particular Cassandra node, given as the number of requests of three classes in that node. On the vertical axis, we show the probability that the system is in the corresponding state. The two models have very similar mean performance metrics such as throughput, response time and CPU utilisation with the maximum error across the three metrics below 12%. The similarity in CPU utilisation is also demonstrated observing the first state $(0, 0, 0)$, which represents the probability to observe the system without any outstanding requests. For this state only, the two probabilities are very similar. However, for the remaining states, the marginal state probabilities are significantly different.

To understand the similarity of these two probabilities, we calculate their divergence using the Kullback-Leibler measure. This measure quantifies similarity between two probabilities distributions (see Section 5.4.2). The closer the returned value is to 0, the more similar the two probabilities are. However, if the value is above 1, the algorithm suggests that the two probabilities are not very similar. In this case, the algorithm returns a value of 2.41, suggesting that even if the predicted performance metrics are close to the real system ones, the underpinning state dynamics in the performance model not resemble the real system dynamics. Our goal in the rest of the chapter is to propose a method that penalizes the estimation of demands that result in high divergence scores between predicted and measured state probabilities.

| Notation | Description |
|---|---|
| $M$ | Number of nodes in the model. |
| $K$ | Number of queueing stations. |
| $R$ | Number of job classes. |
| $C$ | Number of job chains. |
| $N_r$ | Number of jobs in class $r$, $N = \sum_r N_r$ |
| $\theta_{kr}$ | Service demand of class $r$ at node $k$. |
| $\sigma_r$ | Think time of class-$r$, $\sigma_r = \sum_{k=K+1}^{M} \Theta_{kr}$ |
| $c_i$ | Number of servers at node $i$ |
| $s_{ir}$ | Service time at node $i$ for jobs in class $r$ |
| $e_{ir}$ | Visit ratio at station $i$ for jobs in class $r$ |

Table 5.1: Summary of main notation for queueing network models

## 5.3 Efficient marginal probabilities calculation

In this section, we present the algorithm we developed to efficiently calculate the marginal probability for BCMP model that use class-switches. The formulas are based for the product form solver we presented in Section 2.3. To facilitate the reader, in Table 5.1 we report the main notations used to describe the algorithm.

In the evaluation section, we have the necessity to often compute the marginal probability for a given a set of jobs $\boldsymbol{n}_i = (n_{i1}, n_{i2}, \ldots, n_{iR})$ that resides at station $i$. However, because the model is solved by aggregating classes into chains, the underpinning state space describes the state of station $i$ as $\hat{\boldsymbol{n}}_i = (n_{i1}, n_{i2}, \ldots, n_{iC})$, where $C$ is the number of chains, making it difficult to recover per-class metrics.

In [Zah79, p. 110], the authors observe that probability of a particular mix of jobs $\boldsymbol{n} = (n_1, \ldots, n_R)$ being active in the system is the ratio of the normalising constant of a closed model with fixed class populations $\boldsymbol{n}$ divided by the normalising constant of the model with class-switching. Because the factors $f_i(\boldsymbol{n}_i)$ in the product-form solution can themselves be regarded as normalising constants for degenerate models with a single station and population $\boldsymbol{n}_i$, one readily concludes that the mix probability derived in [Zah79] can be used to establish

the per-class population at a node in the aggregated model. This leads to the expression

$$\pi(\boldsymbol{n}_i) = \frac{n_i!}{\prod_{u=1}^{n_i} \min(u, c_i)} \prod_{r=1}^{R} \frac{\theta_{ir}^{n_{ir}}}{n_{ir}!} \frac{G_{\hat{\boldsymbol{\Theta}}}^{-i}(\hat{\boldsymbol{N}} - \hat{\boldsymbol{n}}_i)}{G_{\hat{\boldsymbol{\Theta}}}(\hat{\boldsymbol{N}})} \tag{5.1}$$

where $G_{\hat{\boldsymbol{\Theta}}}^{-i}$ is the normalising constant for the sub-network composed by all stations except station $i$, and $\hat{\boldsymbol{n}}_i$ is obtained from $\boldsymbol{n}_i$ by summing classes that belong to the same chain. Note that both $G_{\hat{\boldsymbol{\Theta}}}$ and $G_{\hat{\boldsymbol{\Theta}}}^{-i}$ are defined for the aggregate model, and thus are computed using the aggregate population vector $\hat{\boldsymbol{N}}$ and the matrix of the aggregate demands $\hat{\boldsymbol{\Theta}}$.

To the best of our knowledge, (5.1) has not appeared before in the literature. This expression paves the way to define a method for estimating service demands in class-switching models, which is presented in the next section.

## 5.4 Estimation Algorithm

In this section, we describe our algorithm for demand estimation. Differently from other algorithms that look into aggregated performance metrics such as system throughput, system response time or CPU utilisation (as described in Section 2.4), the SD algorithm minimises the divergence between the state probability distribution of the real system from the one generated by the model under test. The algorithm is defined in the following section and then we present some divergence measures that can be used to compare the two state probability distributions.

### 5.4.1 SD Algorithm

To be able to work, SD requires a model, provided by the user, of the system under test (SUT). To reduce the execution time of the method, we consider in this work only models that comply with the BCMP theorem, as described in Section 2.3.2, and denote this model by $\mathcal{M} \equiv \mathcal{M}(\boldsymbol{N}, \boldsymbol{\Theta})$. However, the SD algorithm is in principle applicable to any other model from which the state probability distribution can be computed fast enough to apply computational optimisation methods.

Let $\boldsymbol{n}^s = (\boldsymbol{n}_1^s, \ldots, \boldsymbol{n}_M^s)$ denote the $s$-th state sample ($s = 1, \ldots, S$) obtained from the SUT and defined such that $\boldsymbol{n}_i^s = (n_{i1}^s, \ldots, n_{iR}^s)$ is the state of station $i$. The principle underlying SD is to minimise the divergence between observed and predicted *marginal* state probabilities $\pi(\boldsymbol{n}_i^s)$ for the SUT, for all stations $i$ and samples $s$. We denote the empirical and model-based marginal probability distributions with $P$ and $Q$, respectively. That is, $P$ consists of all the marginal probabilities $\pi(\boldsymbol{n}_i^s)$ for every node $i$ and state sample $\boldsymbol{n}^s$ obtained from measurements on the SUT; similarly, $Q$ consists of the corresponding marginal probabilities computed using the model $\mathcal{M}(\boldsymbol{N}, \boldsymbol{\Theta})$.

A conceptual difficulty arising upon optimising state divergences is that this metric does not explicitly consider mean performance metrics, even though these are the typical metrics used once the model is fully parametrized. We propose to address this issue by constraining the divergence minimisation to return predictions for mean performance metrics within a tolerance. The performance metrics considered include, but are not limited to, the system throughput ($X$), system response time ($R$) and the resource utilisation ($U$). Such metrics can be readily obtained in SUT using conventional monitoring tools and in the model using known expressions available for BCMP networks [LZGS84]. To differentiate the real and model-based values, we denote the former for example as $\widetilde{X}$ and the latter as $X$.

Let $D_*(P, Q) \geq 0$ be a generic non-negative f-divergence function used to compare the two probability distributions. Based on the previous considerations, we define the SD optimisation problem as follows:

$$\begin{aligned} SD : \min_{\boldsymbol{\Theta}} \quad & D_*(P, Q) \\ \text{subject to} \quad & |\widetilde{X} - X| \leq \delta \cdot \widetilde{X} \\ & |\widetilde{R} - R| \leq \delta \cdot \widetilde{R} \\ & |\widetilde{U} - U| \leq \delta \cdot \widetilde{U} \\ & 0 \leq \theta_{ir} \leq \widetilde{R} \end{aligned} \tag{5.2}$$

where all the variables of the demand vector ($\boldsymbol{\Theta}$) have, as upper bound, the average response time measured on the real system during the experiment. On the other hand, $\delta \geq 0$ is a tolerance on the maximum relative error on the model mean performance predictions.

Different f-divergence measures can be used with this optimisation problem. In the following section, we present the most popular algorithms and their properties.

### 5.4.2 Divergence measures

In this section, different f-divergence measures are presented, namely Bhattacharyya, Hellinger, Jensen-Shannon and Kullback-Leibler. These measures are divided here into two groups, distance functions and divergence measures. The difference between them is represented by the symmetric property. The distance is a *symmetric* function for which the order of the input probabilities does not change the output results; conversely, divergence measures are sensitive to the order of the input parameters. Another important difference between these measures is the output bound range that can be limited in a range, usually between 0 and 1, or go to infinite. The algorithm to use for a target system can depend on the measure considered and the bound of the results.

#### Bhattacharyya and Hellinger distance functions

The Bhattacharyya (BC) distance is a classical distance function [Bha43, Kai67] and it is used in several fields such as feature extraction [CL03], image processing [GRD04], speaker recognition [YLL09], etc . It is defined as

$$D_{BC}(P||Q) = -\log BC \tag{5.3}$$

where $BC$ is the Bhattacharyya coefficient that for discrete distribution. It is defined as

$$BC = \sum_{x \in X} \sqrt{P(x)Q(x)} \tag{5.4}$$

The Bhattacharyya coefficient can be connected to the geometric mean between the two points. Since the sum of all the probabilities is 1, the Bhattacharyya coefficient lies between $0 \leq BC \leq$

1. Due to the logarithm, the Bhattacharyya distance $D_{BC}$ is consequently defined between $0 \leq D_{BC} \leq \infty$. So, the Bhattacharyya measure is symmetric, unbounded and returns always positive values.

The Bhattacharyya coefficient ($BC$) is also used to define the Hellinger distance (HE). HE is defined as [Hel09, Nik01, SV16]

$$D_{HE}(P||Q) = \sqrt{1 - BC} \tag{5.5}$$

It lies between $0 \leq D_{HE} \leq 1$ and it is a linear function, differently from the Bhattacharyya that has a logarithm behaviours. This distance is frequently applied in multi-criteria decision maker [LK14], feature extraction [CLC+15] and independent component analysis [JWY15].

**Kullback-Leibler and Jensen-Shannon divergence**

Differently from a distance metric, the Kullback-Leibler (KL) divergence (or relative entropy) is a generalisation of the Rényi entropies [Rén61]. It is an asymmetric function, so output of the function of $P$ and $Q$ is not equal to the results of $Q$ and $P$.

Since we use this metric in a discrete space to measure the divergence between two marginal probabilities, here we consider only the Kullback-Leibler divergence discrete formulas. It is defined as

$$D_{KL}(P||Q) = \sum_{x \in X} P(x) \cdot \log \frac{P(x)}{Q(x)} \tag{5.6}$$

The Kullback-Leibler divergence (KL) can be also expressed as the expectation of the log difference between the two probability distributions. The expected value of a discrete function is equal to the weighted sum of all possible values.

In other words, the closer the KL divergence value is to 0 the more similar the two probability distributions are. On the other hand, if the value is greater or equal to 1, several differences

between the two probability distributions are present.

The Jensen-Shannon (JS) divergence is based on the Kullback-Leibler measure. It differs from the latter for being a symmetric function, bounded within a finite range. The JS divergence is defined as

$$D_{JS}(P||Q) = \frac{1}{2}D_{KL}(P||M) + \frac{1}{2}D_{KL}(Q||M) \tag{5.7}$$

where M is the average between the two distributions and it is defined as

$$M(P||Q) = \frac{1}{2}(P + Q) \tag{5.8}$$

The Jensen-Shannon divergence is bounded between $0 \leq D_{JS} \leq 1$

## 5.5 Evaluation

In this section, we present the evaluation of our SD algorithm. We start our investigation with a case study base on Apache Cassandra. From a real system, we record some performance metrics of the overall system and of each node as well as some network communication across the nodes. We use this sniffed communication to construct the system state and gather the marginal probability distribution of the real system. In addition, we also build a queuing network representation of the real system to use with our optimisation algorithm. Using these elements, we perform a sensitivity analysis across several factors such as optimisation algorithms, state strategies, divergence measures and number of state to use as searching space. We then present the performance of our model based on the demands provided by the SD algorithm demonstrating that the algorithm is able to detect the set of demands that are able to approximate best performance of the real system and so to reproduce the system behaviour comparing on the marginal state probabilities of the two systems.

We further investigate the performance of our algorithm against a set of 100 random models.

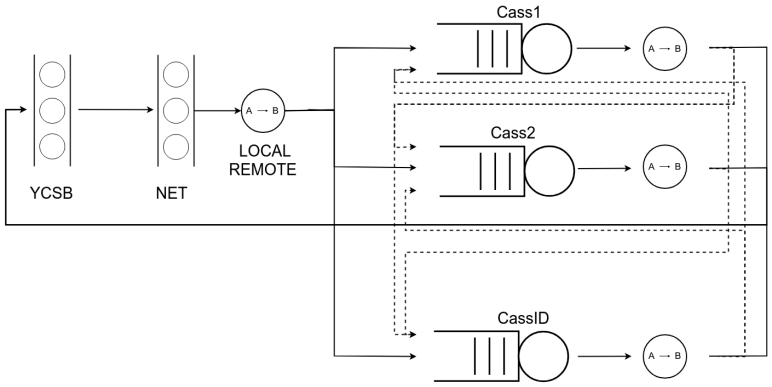



Figure 5.2: Simplified Cassandra model.

We demonstrate that, using more accurate information that presents also less noise, the SD algorithm is able to infer the demands with a lower error and quicker.

### 5.5.1 Cassandra Simplified Model

In this section, we illustrate the Cassandra model used for evaluating the SD algorithm. This is a simplified version of the model presented in [DCS17]. With the aim to reduce the model complexity and the system state space, we have developed a model able to support only the Consistency Level *ONE*. We represent with a single queue each Cassandra node, as shown in Figure 5.2. Each Cassandra node uses the processor sharing (PS) scheduling policy. The disk and network stations used in [DCS17], have been grouped in a single infinite queue (or infinite server), called "Net" positioned right after the workload generator, assuming that the requests have few variances between them and so the demands of these resources can be group together in a single queue. Differently, if significant variance in terms of size of the requests is observed on the system, this model could not be able to represent the performance of the real system correctly. Moreover, the workload generator has also been modelled as an infinite server. Similarly to [DCS17], all the Cassandra nodes that compose the cluster are interconnected with each other to retrieve the data that is necessary to complete the client requests. To communicate with the workload generator, each node uses a separate connection. Moreover, due to the simplified structure of the model and the usage of only the Consistency Level ONE, we are able to remove the fork-join elements from the previous model and reduce the classes to 4. During

| Class | Description |
|---|---|
| local-pars | Local request operations. |
| remote-pars | Remote request parsing operations. |
| remote-ID | It can represent a remote ending operations (remote-end) or data request(remote-incoming). |

Table 5.2: Description of the classes used for in simplified Cassandra model

the execution, a query needs to perform several operations before returning to the workload generator and terminating the query execution. The classes are:

- *local*: accomplishes all the parsing operations for a local query. These operations involve the interpretation of the received query, identify the location of the data on its local disk and the ending operations of the query.
- *remote-pars*: accomplishes all the parsing operations for a remote query and generates the remote-incoming request to one of the clients that holds a copy of the data.
- *remote-end*: the ending operations of a remote request are performed before returning the data to the client.
- *remote-incoming-ID*: the request generated by the node to gathers the data to another node. The node to contact is defined with "ID". So multiple classes like this one needs to be generated based on the number of Cassandra nodes in the model.

To reduce the number of classes in the model, we aggregate together the *remote-end* and the *remote-incoming* classes in the *remote-ID* class as reported in Table 5.2. The model differentiate the demands to use in each case based on the Cassandra-*ID*. In fact, if this class is executed on the same ID node as the station, the *remote-end* operation is performed otherwise, this class represents a *remote-incoming*.

### Model Flow

As in the complete Cassandra model in [DCS17], the node representing YCSB is set to be the reference station and represents the workload generator. All the requests generated by YCSB

start in the local class. After traversing a delay station to describe the network latency (Net), part of the jobs changes class in remote-pars and are sent randomly to one of the Cassandra nodes. The number of requests that change class depends on the total number of nodes that compose the cluster and the replication factor used. It is defined as $(N - RF)/N$ [DCS17]. In case of a local request, the query is processed by the node and then returns directly to the workload generator.

On the other hand, a remote-pars request is processed by the receiving node and then switches in the remote-ID class to be delivered to another node. The job is processed as a remote incoming request and, when terminates, it is sent back to the previous node before returning to the workload generator as a local class.

**State conversion of the real system**

Here, we present how the state are gathered by the real Cassandra system analysing the network traffic. Since we do not want to change the Cassandra source code or its behaviour to gather more accurate information, the state of this system are obtained only analysing the source and destination of the packets. Analysing the IP and port of the source and destination of the packets, we are able to identify the three classes that are executed on a target node:

1. *new-query*: all the new queries that come from the workload generator IP to the target node. For these requests, we are not able to identify if a query is local or remote;
2. *remote-end*: represents the last operations of a remote request. The remote request is identified if a *remote-incoming* request (as defined in the model) has been observed on the network;
3. *request-incoming*: represents the requests issued by the other nodes to the target node of the cluster to gather the data regarding a specified record.

To convert the model state to the Cassandra one before applying the divergence measure, the model marginal probability is transformed as:

| Resource | Description |
|---|---|
| Cloud provider | Microsoft Azure (IaaS). |
| VM type | Standard_D2s_v3. |
| CPU | 2 vCPUs. |
| Memory | 8 GBs. |
| Disk size | OS 30GB, Data 80GB. |
| Disk type | Premium (SSD). |
| Cassandra nodes | 3 (version Apache Cassandra 3.11). |
| YCSB nodes | 1 (version YCSB 0.12.0). |
| YCSB workload | Workload C (100% read). |

Table 5.3: Testbed details.

1. *new-query*: we search all the possible combination of the *local* and *remote-pars* that summed together obtained that value;

2. *request-incoming*: all the *remote-incoming* requests from the other nodes are summed together;

3. *remote-end*: is taken the same value as the model.

Thanks to this transformation, the two marginal probability distributions can be compared using one of the divergence measures presented in section 5.4.2.

### 5.5.2 Experiment settings

In order to evaluate the SD method, we have collected an empirical trace from a Cassandra ring deployed on Microsoft Azure. As the validation is done for illustrative purposes, a small cluster composed by 3 Cassandra Virtual Machines (VMs) has been set up. However, the results are not expected to significantly depend on the number of nodes. Details of the testbed are reported in Table 5.3.

The version of Cassandra installed in the testbed is 3.11. As workload generator, we have decided to install Yahoo! Cloud System Benchmark (YCSB) tool [CST+10] on one additional VM. Since the read operation in Cassandra is the most complex and thus the most challenging to characterise, we decided to focus mainly on a read-only workload.

We warm up the system running for 10 minutes the cluster with the predefined number of clients. Keeping the workload generator running, we start to record all the Cassandra traffic communication of the target node with the YCSB and with the other clients using *tcpdump* tool [Tcp18]. Different time lengths for sniffing traffic have been tested and we decided to use one minute since the marginal probability is stable. We do not use longer periods because the sniffed data grows very quickly so more data needs to be stored and more time is required to process them. We develop some Python scripts to analyse the recorded traffic. Looking at the source and destination of the connections, the scripts are able to detect the state of the system at each stage and so construct the marginal probability of the system for the recorded period of time.

### 5.5.3 Minimization algorithm settings

We have first tested the SD demand estimation approach using four different algorithms for the optimisation of the underpinning non-linear program, namely Fmincon [BGN00, WMNO06], GlobalSearch (GS) [ULP+07], MultiStart (MS) and Genetic Algorithm (GA) [GH88, CGT97, Mit98], which are all available in MATLAB version R2018b. More details about these optimisation algorithms can be find in Section 2.4.4.

For our evaluation, we have used for all the algorithms the same objective function, non-linear constraints and bound $[10^{-10}, RT_{system}]$ for all the decisions variables (i.e., service demands), where $RT_{system}$ is the average response time observed by the workload generator. In the case of the Fmincon algorithm, by default the starting point for all the variables are taken randomly inside the range. Since this parameter can influence the final results, to achieve repeatable outcome, we set all the values to half of the system response time value. Of course, this can influence the final results but this is necessary to be able to compare the algorithm through the different cases. Regarding the non-linear constraints, we limit the evaluation only to the overall system throughput to make sure that the set of testing demands has a relative error ($\delta$) of less than 20% from the real one.

The Cassandra queueing model uses 5 random variables representing the demand due to the

network delay and the demands for the four classes of the model (local, remote-pars, remote-end and remote-incoming), which are identical at all nodes. By default, the NC solver available in the LINE solver [PC17, Cas19] is used to analyse and retrieve performance and marginal probabilities of the model under test, using an implementation of the expressions in Section 2.3.2. The two probability distributions are then compared using one of the divergence measures presented in Section 5.4.2. Concerning the workload generator, its demand is estimated considering the CPU time that YCSB spent into the system divided by the number of requests generated in that period of time.

### 5.5.4 Sensitivity analysis

In this subsection, we perform a sensitivity analysis of our results with respect to the state space size sampled from the real system and the choice of divergence measures.

#### State Space

Some considerations about the state space used with the SD algorithm are presented in this section. We start presenting how quickly the number of states grows increasing the number of clients in the system and then we present the impact of three state spaces on the performance of the SD algorithm.

As described in Section 5.4.1, the SD algorithm considers all the states that have been observed in the real system during the monitoring period. When the number of clients in the system is small, the state space can be exhaustively observed if the monitoring period is long enough. However, when the number of clients grows, the state space size grows exponentially according to the formula

$$\boldsymbol{S} = 1 + \sum_{n=0}^{N} \binom{n + L - 1}{L - 1} \tag{5.9}$$

where $N$ is the total number of jobs into the system and $L$ is the product between the number of classes of the chain and the number of stations in the system or model. Since our target node is a single station with 3 classes, for our calculation the value of $L$ is equal to 3.

| # Clients | Observed | Actual size |
|---|---|---|
| 1 | 4 | 4 (100%) |
| 10 | 234 | 286 (81.8%) |
| 30 | 2939 | 4960 (59.2%) |
| 50 | 10302 | 23426 (43.9%) |

Table 5.4: How the total number of states grows with the number of clients.

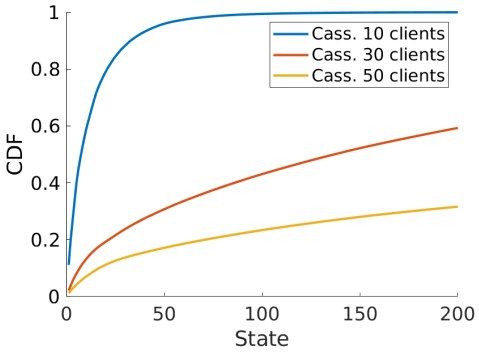
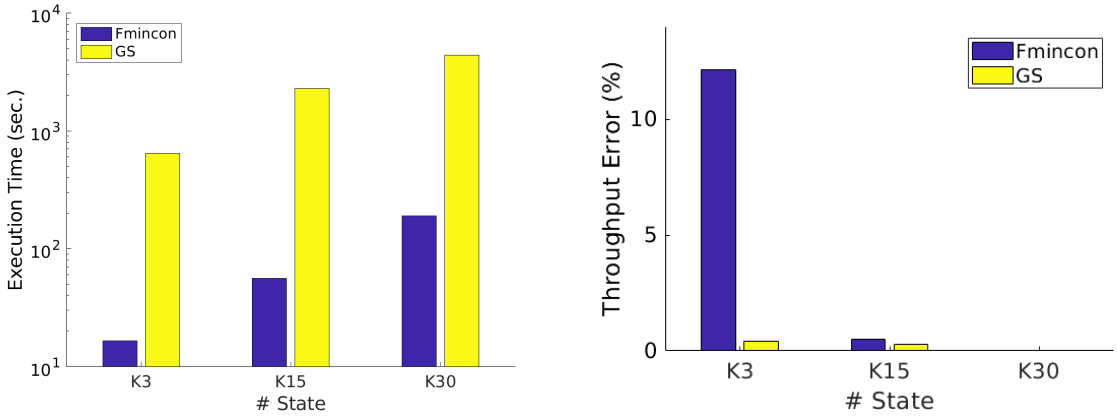



Figure 5.3: CDF for the first 200 state of the marginal state probability.

As reported in Table 5.4, for Cassandra the state space is composed of 4 states with 1 client, but grows to 286 states with 10 clients of which only 234 states were observed during a reference experiment of one minute. Since it is very expensive, or even intractable, to compute marginal probabilities for the complete state space, we have decided to limit the number of states to a maximum of $k$ states. That is, we obtain marginal probabilities from the system for $k$ states only. All the remaining states are aggregated into a single unobserved state to complete the state probability distribution. We evaluate the SD algorithm using $k = \{3, 15, 30\}$; we shall equivalently refer to these three cases as $K3$, $K15$, $K30$. The states to include into the sample space are selected based on the $k$ value.

The state selection is performed by dividing the probability distribution into three sections (high, medium, low) and take an equal number of states from each group. However, these three sections are applied to the state with a probability higher than 0.00001 only because the marginal state probability trend decreases quite quickly. To demonstrate how fast the marginal state probability decreases, in Figure 5.3, we show the CDF of the first 200 states using different number of clients. The figure demonstrates also that increasing the number of clients into the system the probability values of the first few state decrease.

To demonstrate the impact on the SD execution time of these three different state spaces, we have run an experiment with 10 clients in the system. We decided to use 10 clients because, with more clients, the search takes significantly more time to complete. Differently, with less clients, all the state are considered and so the different strategies cannot be compared. Moreover,




(a) Execution time

(b) Throughput error

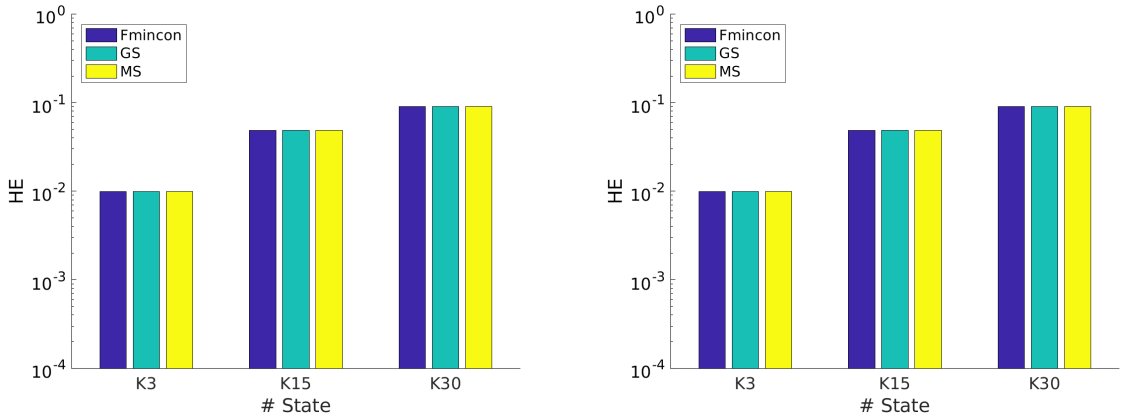
Figure 5.4: Execution time and throughput error in relation to the number of elements in the search state space.

for this evaluation only, we take in consideration two optimisation algorithms, Fmincon and GlobalSearch, to better understand the impact of the searching space with a simple and a more complex algorithm.

The difference of the execution time is reported in Figure 5.4a. It is clear that as the number of elements inside the searching state space grows, the time required by the Fmincon and GlobalSearch algorithms to complete grows linearly. This is caused by two main factors: with a bigger searching state space, the optimisation algorithm computes more steps before completing and the time required by the solver to analyse and retrieve the state probability is larger. This effect is clearly emphasised looking at Fmincon optimisation algorithm, where the execution time grows exponentially with the number of elements in the searching state space. On the other hand, using a larger searching state space, the final set of demands returned by the algorithms presents a lower percentage of error on the system throughput and it is more reliable than a small one, as reported in Figure 5.4b.

### State Strategy

Two possible strategies to calculate the marginal probability from the real system are presented here. These two strategies are defined as:




(a) Frequent (b) Largest

Figure 5.5: Comparison between Frequency and Largest strategies with 10 clients and using HE.

- Largest: considers the top-$k$ most probable states, as defined by the amount of time spent in each state.

- Frequent: considers the top-$k$ most occurrent state of the system without considering the time spent on each state. For the evaluation of this strategy, the *arrival theorem* [LR80, SM81, LZGS84] has been applied.

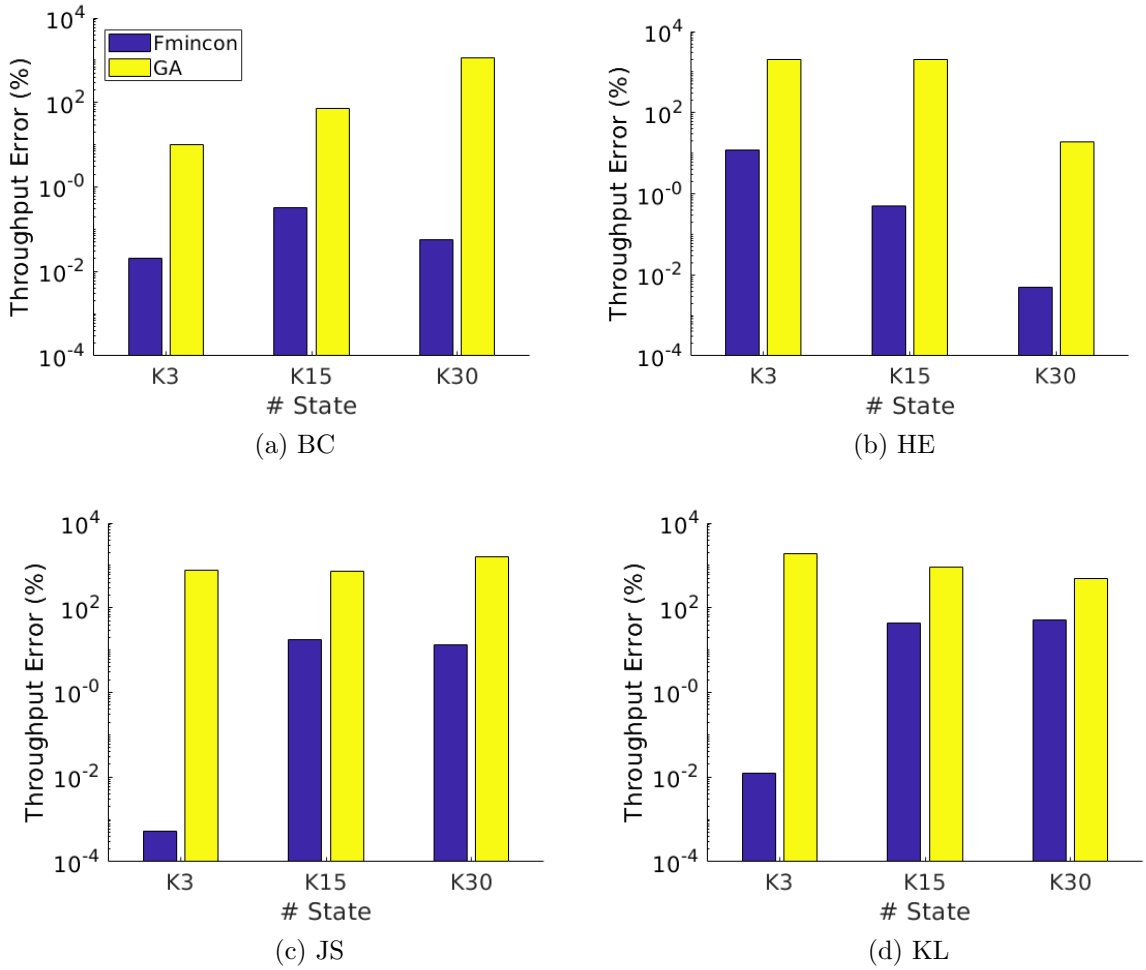
To understand which of these two state strategies is able to perform better and so improving the algorithm performance, we run an experiment with 10 clients using different optimisation algorithms. The final values of the SD algorithm using HE distance is reported in Figure 5.5. The figure reports that the two strategies are very similar and no significant difference is presented between them. In addition, increasing the number of state in the searching state space, both strategies increase similarly the value returned by the selected divergence measure. Since the largest strategy presents a slightly lower mean error we decided to use this strategy for the rest of the experiments. However, since the two strategies are very similar, both can be used for the demand estimation with our model.

**Divergence measures**

In this section of the sensitivity analysis, we present the results of a set of experiments necessary to select the right combination of optimisation and divergence measure able to provide stable results at the end of the SD algorithm execution. We perform our investigation in two steps. We first compare two different optimisation algorithms: one based on derivative (Fmincon) and one evolutionary algorithm (GA). Since derivative approach performed better on our initial evaluation, we investigate if more complex and robust optimisation algorithms can improve the performance of SD and which divergence algorithms is capable of achieving the best results. Ideally, this combination needs to provide an algorithm that it is reliable, so able to explore all the space generated by this model, and able to reduce the performance error of the algorithm increasing the state space used.

As first step of our investigation, we check which optimisation algorithm performs better in this context. We take into consideration the case with 10 clients using the Largest strategy. To have a broader view, we compare these algorithms using 3 different sample space sizes ($K3, K15, K30$) and dividing the comparison of the optimisation algorithms in two groups. We first compare the Fmincon as derivative approach against the GA that is an evolutionary algorithm. For these set of experiments, we use the default Matlab settings for these two algorithms. The only change we made is to set as starting point for the Fmincon algorithm half of the average response of the queries.

In Figure 5.6 the results of the comparison between Fmincon and Genetic Algorithm are presented. It is visible that the derivative-based algorithm outperforms GA in any of the analysed situations. In most of the cases, the difference between the two algorithms is in the order of one or more magnitudes. This can be caused by the genetic algorithm settings that we used for these experiments. Usually two parameters can influence the GA performance. The first parameter is the *crossover* parameter, which defines how to combine the new generation based on the parents' information. For non-linear optimisation problem three main approaches are available in Matlab. We decided to use the default one (*crossoverscattered*) because it does not take sequences of values from the parents but it decides randomly for each parameter



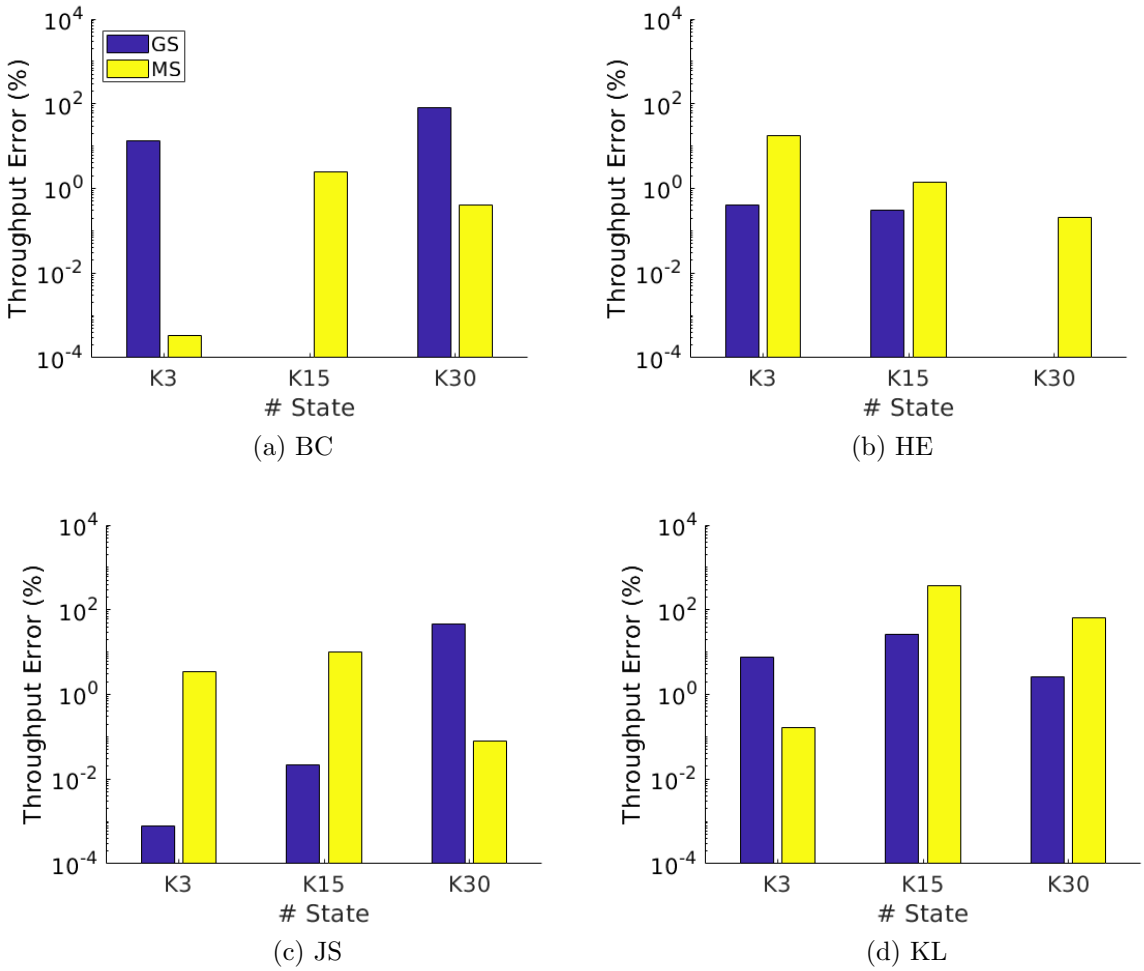

(a) BC (b) HE

(c) JS (d) KL

Figure 5.6: Divergence comparison with 10 clients and using Largest strategy with Fmincon and GA.

from which parent inherit the value. The second parameter is the *mutation* that can influence the new generation. The algorithm randomly changes few individuals of the population to create mutation children. This enables the genetic algorithm to search a broader space and create diversity. The choice to use the default genetic algorithm setting has been preferred for simplicity and reproducibility reasons. However, this may not be sufficient to draw results about the effectiveness of genetic algorithms for divergence estimation in general, as the latter would require a more extensive sensitivity analysis for optimal parameter tuning of the specific algorithm of choice, which will be investigate as future work.

Focusing the attention on Fmincon, the BC and HE divergence measure perform better than the other two algorithms with a throughput error significantly lower than 10%, in most of the cases. Furthermore, HE presents the lowest error value among the K30 and its error is reduced



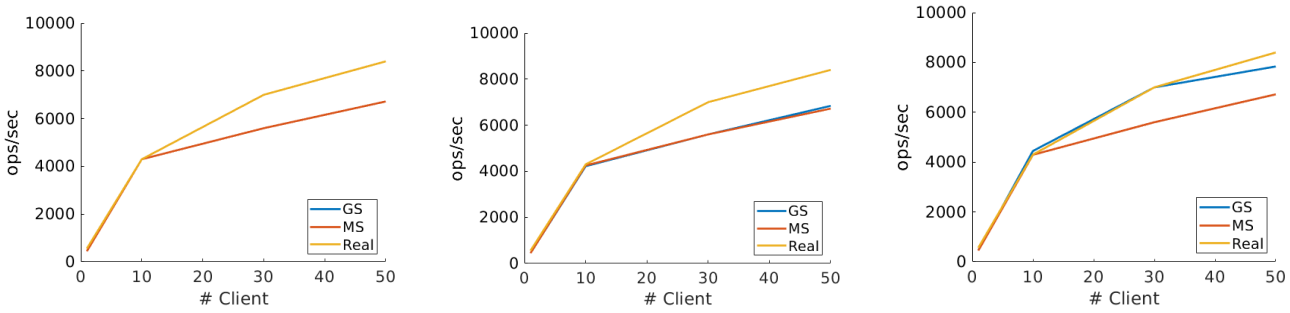

(a) BC (b) HE

(c) JS (d) KL

Figure 5.7: Divergence comparison with 10 clients and using Largest strategy with GS and MS.

linearly with the growing number of states as part of the searching space.

Since the derivative-based algorithm outperforms the genetic one, we evaluate if more robust derivative optimisation algorithms such as GS and MS are able to perform better. Figure 5.7 shows that these algorithms reduce further the throughput error and, in few cases, the errors are negligible. Considering now the divergence measure, we notice that the HE measures using these two optimisation algorithms presents the property that we are looking for the SD algorithm. In fact, increasing the searching space, it reduces the error gradually. Moreover, when we use GS with HE and $K30$ as state space, the algorithm reaches a negligible throughput error. Different is the behaviour of the other divergence measures where the error is not constant or decreases using bigger state searching space, like BC and KL, or where they even increase the error with a larger state space, like Jensen-Shannon using GS.




(a) K3 (b) K15 (c) K30

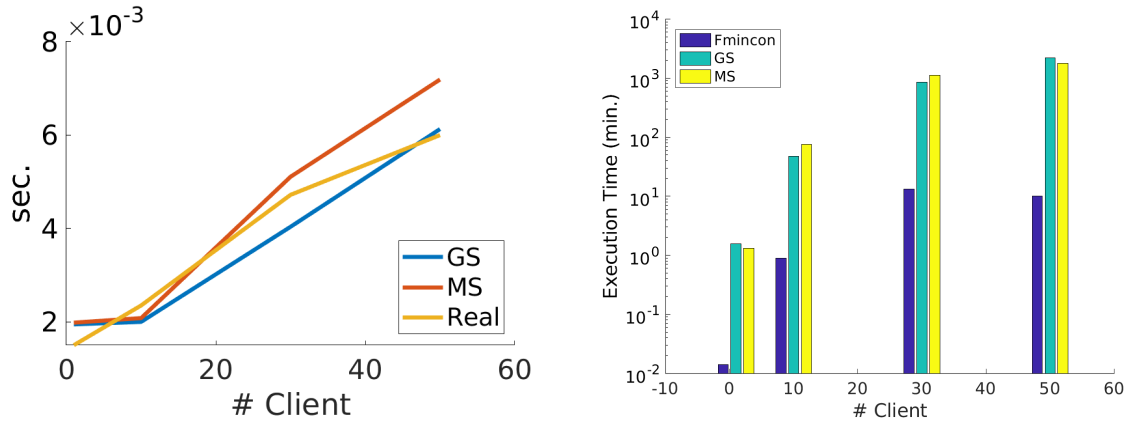
Figure 5.8: Predicted throughput performance using different number of elements in the searching state space.

For these reasons, we have decided to consider HE as the main divergence measure. Differently, as optimisation algorithm, both the GS and MS can be used. However, in some cases, we still considering the Fmincon optimisation algorithm because it is significantly faster than the GS or MS.

### 5.5.5 Cassandra Demand Estimation

We now examine the performance of the SD algorithm on the Apache Cassandra case study. We first present a comparison between performance metrics of the model parameterized with the demands returned by the SD algorithm and the same metrics recorded in the real system. Then, we carry out a comparison between the two state probability distributions.

Figure 5.8 illustrates the throughput prediction of the different searching algorithms with different numbers of elements into the searching space. It is clear that, by increasing the sampled state space utilised during the search, the proposed approaches are able to perform better. The benefits of using a larger state space are shown with $K30$ where the SD algorithm using GS and MS predict, with low error, the system performance. The maximum percentage of error achieved with a highly loaded system is around 6.7% for the GS and 20% for the MS. Regarding Fmincon optimisation algorithm, using $K3$ and $K15$, the results provided at the end of its execution are comparable with the other two algorithms. However, using $K30$ Fmincon is not able to find any acceptable solution with 30 and 50 clients in the system returning a high error




(a) Response Time (b) Execution Time

Figure 5.9: Estimated response time and execution time of the model with $K30$ state.

value. Since this algorithm does not use multiple starting points, it is less robust and it is more difficult to reach a good final value when the data present noise or are not precise enough.

We also analyse the model response time prediction with K30 for the GS and MS algorithms. The results are shown in Figure 5.9a. For GS, the average system response time is close to the real one with an average error of around 14%. Similar is also the MS average error of 17%. As expected the execution time for the SD algorithm is in relation to the optimisation algorithm used, as it has already shown previously in Section 5.5.4. However, also the number of clients and the sample state space used can influences significantly the execution time. In fact, growing the value of these two factors, the solver takes more time to analyse the system to predict some metrics and to retrieve the marginal state probability of the system under test. For this set of experiments, the $K30$ is the one which spends more time to produce a result. In particular, the time that each algorithm spends to complete the execution is reported in Figure 5.9b. The difference in time of Fmincon against GS or MS is relevant with all the analysed cases taking, in most of the cases, at least one order of magnitude less than all the others. However, Fmincon is not always able to find a valid set of demands able to satisfy user constraints. On the other hand, GS and MS use comparable time to complete.

To show further benefits of the SD algorithm, we report in Figure 5.10 the state probability comparison between the real system and the model with 10 clients using $K30$ as sampled state space. It is possible to see that, differently from the one figure presented in Section 5.2, the two

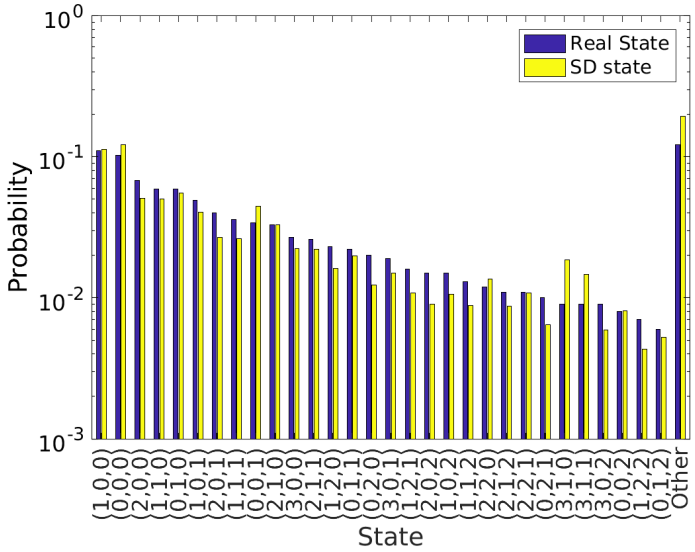



Figure 5.10: State probability distribution for the model with $K30$ state and 10 clients.

distributions are very similar one each other and the distance value, using the HE measure, is 0.0938. This underlines that the set of demands returned by the SD algorithm is able not only to predict the performance metric of the system, but also to generate demands that correctly capture the real system dynamics.

## 5.6 Random Models

To further investigate the robustness and performance of SD, we run a set of random experiments to analyse the sensitivity of the result to different demands, $\delta$ parameter value, and method execution times.

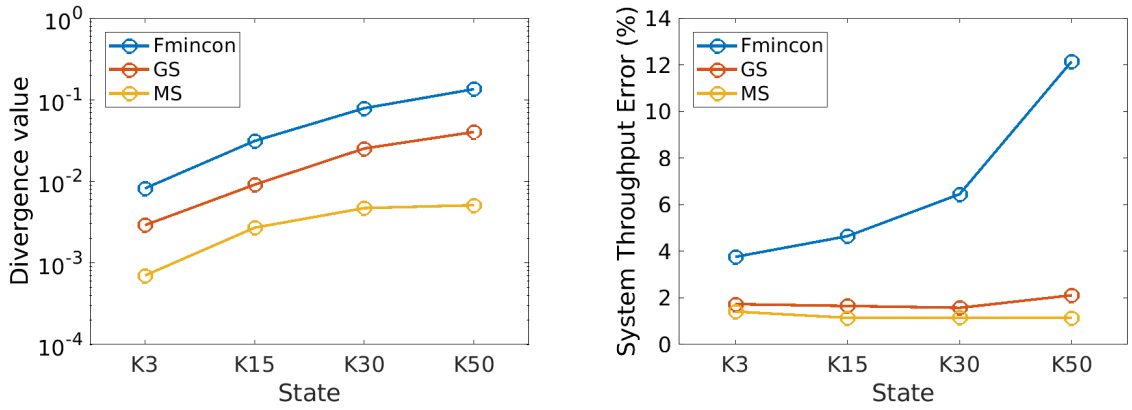
To perform these experiments, we have created 100 sets of random demands, each one containing four demands, one for each service class, in the range between 0.00001 to 0.1. Each set of demands is then set into the model presented in Section 5.5.1 to calculate the marginal state probability of the model and average performance metrics such as throughput, response time and CPU utilisation of one node. These metrics are obtained by analysing the model using the formulas presented in Section 5.3. For each random model, we then run the SD algorithm using different values of $\delta$, chosen in the set $(0.2, 0.5, 0.7, 1)$, with and without non-linear constraints,

| | | $\delta = 0.2$ | $\delta = 0.5$ | $\delta = 0.7$ | $\delta = 1$ | Without |
|---|---|---|---|---|---|---|
| **Fmincon** | **K3** | 215.21% | 207.85% | 194.50% | 241.95% | 54.10% |
| | **K15** | 237.82% | 213.45% | 206.21% | 255.92% | 6.12% |
| | **K30** | 334.69% | 271.73% | 240.30% | 303.68% | 3.27% |
| | **K50** | 490.13% | 432.53% | 317.74% | 427.76% | 1.12% |
| **GS** | **K3** | 72.39% | 66.57% | 64.53% | 74.69% | 54.23% |
| | **K15** | 33.22% | 25.54% | 11.06% | 11.24% | 5.90% |
| | **K30** | 33.98% | 30.08% | 18.56% | 23.91% | 3.96% |
| | **K50** | 61.52% | 48.13% | 26.40% | 31.91% | 1.03% |
| **MS** | **K3** | 50.36% | 49.94% | 51.80% | 50.99% | 53.44% |
| | **K15** | 7.11% | 5.16% | 5.69% | 6.30% | 5.96% |
| | **K30** | 4.42% | 2.40% | 3.58% | 3.34% | 3.48% |
| | **K50** | 1.76% | 0.78% | 0.95% | 0.71% | 1.06% |

Table 5.5: Average percentage of error of the founded demands with SD algorithm using the random models.

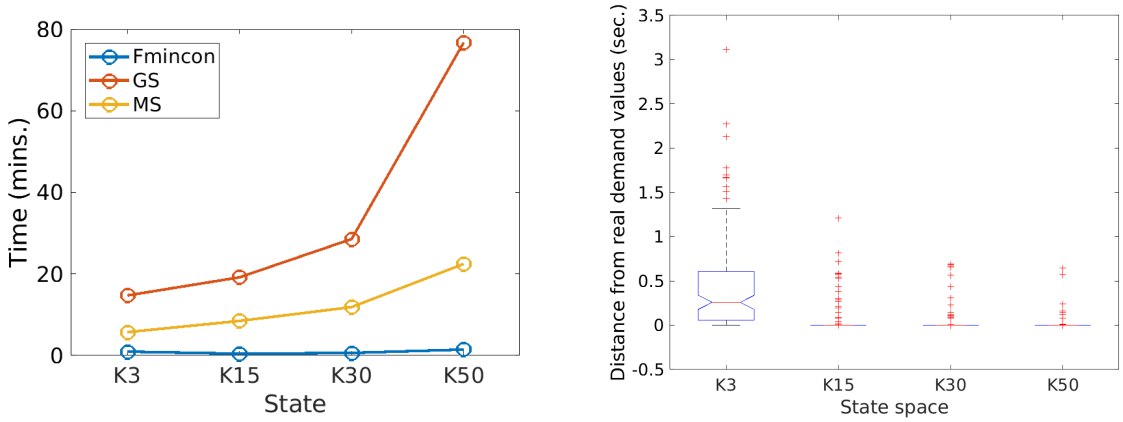
different sample state space sizes ($K3$, $K15$, $K30$, $K50$), and with Fmincon, GS and MS searching algorithms. We run the random models with HE and 10 clients. As for the previous experiments, the searching space for the demands is limited by the average response time of the system and, for Fmincon, the starting point for all the demands is set to half of the system response time. In these experiments, we keep in consideration the Fmincon as reference point for the other algorithms but we do not consider this algorithm robust enough to be used for real demand estimation as it has been demonstrated in the previous section.

Table 5.5 presents the mean percentage of error collected between the real and the found demands using different algorithms, search space and $\delta$. It is clear that the MS algorithm outperforms the other two algorithms. In particular, for all the sampled state spaces bigger than $K3$ and under varying $\delta$ values, the algorithms are able to find with a very low percentage of error the real set demands. Even in the case with $K3$ the algorithm is able to find good demands compared to the other algorithms using the same state space. However, the absence of state information makes difficult for the algorithm to find better demands. On the other hand, the other two algorithms present a quite high percentage of error in most of the cases compared to the MS. The GS algorithm is able to perform better than Fmincon since the multi starting point gives the opportunity to the searching algorithm to reach closer to the real demands. So in this case, a more structured and equally distributed starting point approach is able to perform better.




(a) Divergence (HE) (b) System throughput

Figure 5.11: Average Divergence value and System throughput error for Random Model with $\delta = 0.2$.




(a) Mean Execution Time (b) Demands distance

Figure 5.12: a) Average Execution Time for Random Models with $\delta = 0.2$ b) Mean distance between demands of each class compared to the real one using MS and $\delta = 0.2$.

It is also interesting to see the case where the non-linear constrains are included in the search algorithms. In this situation, the algorithms are able to perform similarly to each other and the difference in performance is dictated only by the sampled state space size. Based on these results we have tested if, without non-linear constraints, the demand inference in Cassandra can improve as well or reduce the execution time. However, we noticed that the optimisation algorithms return, in this condition, higher final divergence values and in some case they are not able to converge. For this reason, we do not advise to use this method for a production system.

The different performance of these algorithms are also represented by the HE divergence value

registered by the different algorithms. In Figure 5.11a are presented the mean divergence value of the different searching algorithms with $\delta = 0.20$. With all the state spaces, the MS algorithm reaches HE divergence values significantly lower than the other two searching algorithms. In some cases, such as $K30$ and $K50$, the difference between them is larger than one magnitude indicating that a more structured starting points strategy is able to perform better. This is also reflected in the accuracy of the model performance prediction. Figure 5.11b shows the average system throughput error of the random models. The MS and GS present a very low percentage of error, around 2%, but the MS still performs better in any analysed situations. Differently, Fmincon presents a considerably higher error and it increases with the state space utilised. This underlines that some local minimum, during the searching process, can deviate the Fmincon from the global minimum.

Significantly different is the execution time between these three optimisation algorithms, reported in Figure 5.12a. As expected, the Fmincon algorithm presents the lowest execution time and it does not depend on the number of state in the searching space as the other algorithms. The other two algorithms spend significantly more time before returning the result. In particular, the GS algorithm execution time increases exponentially with the number of state inside the searching space. Also the MS increases its execution time but less than the GS meaning that it needs to perform a lower number of steps before completing its execution.

Finally, in Figure 5.12b, we report the distribution of the absolute error between the demands returned by the MS algorithm using $\delta = 0.20$ and the real demand values. It is clear that increasing the searching space, the number of outliers considerably reduces and the error decreases closer to zero. Only $K3$ presents a more substantial error in the demands values. So, when it is possible, it is advisable to use at least 15 state in the searching space.

## 5.7 Summary and Conclusion

In this chapter, we present a new algorithm, called SD, that is able to infer the demands in distributed cloud applications. The SD algorithm differs from state-of-the-art approaches

mainly for two reasons: firstly, it uses the marginal probability of the real system to perform the inference of the demands rather than average metrics as done in most algorithms. Secondly, users need to provide to the algorithm a representative queueing network model of the system, allowing to capture the entire workflows of the requests.

Through a case study, the sensitive analysis has been conducted using a real Apache Cassandra cluster deployed on the Microsoft Azure cloud. The results shown that the SD algorithm identifies a set of demands able to not only match the average metrics performance, but also the system behaviour represented by the marginal probability of the analysed system.

Further validation has been conducted on a set of random models where the demands have been generated within a specific range. In this case, we have demonstrated that in absence of system noise, the SD algorithm is able to perform well and incur a small error around 2%.

# Chapter 6

# Conclusion

## 6.1 Summary of Thesis Achievements

In this thesis, we have explored new approaches for the performance evaluation and optimisation of NoSQL database systems. The motivation behind this work is to create new tools to better analyse the performance, estimate with accuracy the demands and optimise the running costs using an autoscaling system for this highly distributed system and databases.

To achieve this, in Chapter 3 we have presented a new queueing network model to describe NoSQL databases. The model integrates the key features that this class of databases most frequently implements, such as synchronisation of the query execution between nodes, replication factor and consistency levels. Moreover, the proposed model offers a simple way to configure them based on the real system set up. To test our model, we primarily targeted the Apache Cassandra database because it implements all these features. However, we demonstrate that, with few changes, the model can be used also for other popular NoSQL databases, such as ScyllaDB. We have validated the model on different environments composed on a different number of nodes, consistency levels and replication factors. For each setup, we evaluate the performance of our model to predict the throughput, response time and CPU utilisation with the different number of jobs in the system. For the experiments conducted on the private cloud, we have demonstrated that the model is able to predict with an average error below the 6%

the real throughput system while, using the public cloud, the error is slightly higher. We have then improved the performance of our model on the public cloud estimating the demands in multiple points and fitting them on some mathematical functions. Thanks to the fittings of the demands, we can estimate with more accuracy the performance of the real system reducing, in some cases, to around 5% the average error of the model. Through a case study, we have also demonstrated the utility of queueing networks models for this kind of applications.

In Chapter 4, we have presented a new autoscaling system for NoSQL databases that optimises the running costs of the infrastructure. PAX leverages on an Apache Cassandra mechanism, called hinted handoff, that it is activated when one of the cluster nodes is unavailable. In PAX the hinted handoff mechanism is used to speed up the data synchronisation when a node changes state from dormant to active. To reduce the SLO violation, PAX uses the information provided by the workload profiling and the distribution of partitions to perform a better choice when the system needs to scale. During the evaluation, we demonstrate that PAX is able to work properly with reactive and proactive scaling policies with small differences. We evaluate the performance of this system using different workload trends and, in all the analysed situation, the SLO violation is very contained. The results demonstrate also that PAX can reduce the operation costs up to 60% compared to traditional Apache Cassandra deployments. Furthermore, we investigate the performance of a more complex autoscaling algorithm, called OPAX, that shows that it can help to reduce even more the SLO violation.

As last contribution of this thesis, we have presented a novel demands inference algorithm called SD. This algorithm aims to infer a set of demands that not only is able to predict the system demands but it can also reproduce the real system behaviour. To achieve this goal, instead of using aggregated metrics such as average system throughput or response time, the SD algorithm uses the marginal state probability of the target system. To compare the two marginal state probabilities, the SD algorithm uses one of the divergence measures provided by the information theory. After conducting a sensitivity analysis to identify the most appropriate settings, we evaluate the algorithm on Apache Cassandra. The set of demands estimated by the SD is able to predict with accuracy the system performance and reduce the gap between the marginal probability of the real system and the model. Furthermore, we validate the SD

inference algorithm on 100 randomly generated models.

## 6.2 Future Work

There are many potential points for future work. Here we present several, in our opinion, most impacting and promising enhancements for the performance analyses and optimisation of NoSQL databases.

#### Apache Cassandra queuing network model

As future work, we are considering to expand the support of our model to other NoSQL databases that are not column-oriented or present a different set of factors of the CAP theorem such as MongoDB, HBase, Dynamo, etc. Even if the data flow and the architecture for these databases are different we believe that, with few changes, the model can support them. Moreover, we would like to develop a similar model using the extended queueing networks, possibly Layered Queueing Network (LQNs), in order to progress from a simulation model to an analytic evaluation. This can open the possibility to analyse more computationally-efficient the model and the possibility to apply it to nonlinear optimisation programs which are commonly used for capacity planning under performance and reliability guarantees.

In addition, the LQNs can also allow us to create models with a higher number of nodes that are difficult to build with queueing networks due to the number of classes involved and the time required to solve them.

#### Partition-Aware Autoscaling for the Cassandra NoSQL Database

As future work, we would like to extend our PAX controller to support other metrics as well such as response time, throughput, etc. In particular, response time could be useful to address burstiness in workloads which are not easily recognisable with CPU utilisation.

Another area of interest is the policies that could be implemented to maintain the data synchronisation across the nodes to avoid that the hinted handoff stops to record data after reaching the maximum time allowed (as presented in Section 4.4). Furthermore, these policies can open the possibility to reduce the boot-up time of the machines. A possible policy implementation could be that, when the hints tables have reached a defined hint size threshold, the controller changes the status of the machine to run the data synchronisation consuming all the hint tables for the target node. Other possible solutions could be to activate the machine regularly after a specific time of inactivity or when the cluster is in a quiet moment. Probably, the last policy is the one that affects less the performance of the entire cluster and network in particular during the synchronisation. For all the other policies a balance between the time required to run the data synchronisation and the cost of this operation needs to be defined. This balance can be found restricting the hinted handoff transmission rate parameter that defining the data synchronisation speed and so limit the impact of this operation.

**SD: a Divergence-based Estimation Method for Service Demands**

As future work, we would like to further investigate the SD algorithm behaviour with other different types of applications and other families of queueing network models. In particular, we are interested to analyse the SD performance in models where the nodes are not all identical or in models which represent multi-tears application such as a website composed by the web, application layer and database servers. The analysis of these type of models can severally impact on the execution time of the algorithm since the number of configurations and variables can increase significantly.

To address this problem a possible idea could be the implementation of a hybrid solution composed by a two-steps algorithm. The first step executes a preliminary search that tries to limit the space range for each variable while the second step executes a full search within the limited space range. Another possibility is to collect some data from the model to train a neural network able to predict faster the model performance metric without the necessity to solve it for every optimisation cycle.

# Bibliography


[ABF14] Veronika Abramova, Jorge Bernardino, and Pedro Furtado. Evaluating cassandra scalability with ycsb. In *International Conference on Database and Expert Systems Applications*, pages 199–207. Springer, 2014.

[ABG+13] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE Transactions on Software Engineering*, 39(5):658–683, May 2013.

[ACC+14] Danilo Ardagna, Giuliano Casale, Michele Ciavotta, Juan F. Pérez, and Weikun Wang. Quality-of-service in cloud computing: modeling techniques and their applications. *J. Internet Services and Applications*, 2014.

[AEADE11] Divyakant Agrawal, Amr El Abbadi, Sudipto Das, and Aaron J Elmore. Database scalability, elasticity, and autonomy in the cloud. *DASFAA (1)*, 6587:2–15, 2011.

[AHLJ12] Alexandros Labrinidis, H. V. Jagadish, Alexandros Labrinidis, and H. V. Hv Jagadish. Challenges and opportunities with big data. *Proceedings of the VLDB Endowment*, 5(12):2032–2033, 2012.

[ALS10] Eric Anderson, Xiaozhou Li, and M Shah. What consistency does your key-value store actually provide. *Proceedings of the Sixth international conference on Hot topics in system dependability. USENIX Association*, pages 1–16, 2010.

[AM19] Michael Kaufmann Andreas Meier. *SQL & NoSQL Databases: Models, Languages, Consistency Options and Architectures for Big Data Management*. 2019.

[Apa] Apache cassandra. `http://cassandra.apache.org/`.

[App19] Austin Appleby. Smhasher test suite (murmurhash), 2019.

[ASV13] Ahmad Al-Shishtawy and Vladimir Vlassov. Elastman: Elasticity manager for elastic key-value stores in the cloud. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, CAC '13, pages 7:1–7:10, New York, NY, USA, 2013. ACM.

[Bau93] Falko Bause. Queueing Petri Nets-A formalism for the combined qualitative and quantitative analysis of systems. In *Proceedings of 5th International Workshop on Petri Nets and Performance Models*, number class 1, pages 14–23. IEEE Comput. Soc. Press, 1993.

[BBS+77] Gianfranco Balbo, Steven C Bruell, Herbert D Schwetman, et al. Customer classes and closed network models: A solution technique. In *IFIP Congrss*, pages 559–564. North-Holland Publishing Co., 1977.

[BCH+14] Sean Barker, Yun Chi, Hakan Hacigümüs, Prashant Shenoy, and Emmanuel Cecchet. Shuttledb: Database-aware elasticity in the cloud. In *Proceedings of the 11th International Conference on Autonomic Computing (ICAC 14)*, pages 33–43, Philadelphia, PA, 2014. USENIX Association.

[BCMP75] Forest Baskett, K. Mani Chandy, Richard R. Muntz, and Fernando G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *J. ACM*, 22(2):248–260, April 1975.

[BCS09] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. Jmt: performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.*, 36(4):10–15, 2009.

[Bey11] Mark Beyer. Gartner Says Solving Big Data Challenge Involves More Than Just Managing Volumes of Data. *Gartner. Archived from the original on*, 10, 2011.

[BGDMT06] Gunter Bolch, Stefan Greiner, Hermann De Meer, and Kishor S Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.

[BGI14] Enrico Barbierato, Marco Gribaudo, and Mauro Iacono. Performance evaluation of nosql big-data applications using multi-formalism models. *Future Generation Computer Systems*, 37:345 – 353, 2014.

[BGN00] Richard H Byrd, Jean Charles Gilbert, and Jorge Nocedal. A trust region method based on interior point techniques for nonlinear programming. *Mathematical Programming*, 89(1):149–185, 2000.

[Bha43] Anil Bhattacharyya. On a measure of divergence between two statistical populations defined by their probability distributions. *Bull. Calcutta Math. Soc.*, 35:99–109, 1943.

[BKK09] Fabian Brosig, Samuel Kounev, and Klaus Krogmann. Automated extraction of palladio component models from running enterprise java applications. In *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS '09, pages 10:1–10:10, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[BL13] Russell Bradberry and Eric Lubow. *Practical Cassandra: A Developer's Approach*. Addison-Wesley Professional, 1st edition, 2013.

[BM93] A. Bertozzi and J. Mckenna. Multidimensional residues, generating functions, and their application to queueing networks. *SIAM Review*, 35(2):239–268, 1993.

[BM04] Antonia Bertolino and Raffaela Mirandola. Cb-spe tool: Putting component-based performance engineering into practice. In *Component-Based Software Engineering*, pages 233–248, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[Bre00] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[Bru78] Steven Christopher Bruell. *On Single and Multiple Job Class Queueing Network Models of Computer Systems.* PhD thesis, West Lafayette, IN, USA, 1978. AAI7914877.

[BS78] Y Bard and M Shatzoff. Statistical methods in computer performance analysis. *Current Trends in Programming Methodology*, 3:1–51, 1978.

[BS15] Michael Bar-Sinai. Big Data Technology Literature Review. *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing - STOC '97*, 2013(April 2013):654–663, jun 2015.

[BT11] David Bermbach and Stefan Tai. Eventual consistency. In *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing - MW4SOC '11*, number May, pages 1–6, New York, New York, USA, 2011. ACM Press.

[Cas11] Giuliano Casale. Exact analysis of performance models by the method of moments. *Performance Evaluation*, 68(6):487 – 506, 2011.

[Cas19] G. Casale. Automated multi-paradigm analysis of extended and layered queueing models with line. *ACM/SPEC ICPE*, 2019.

[Cat11] Rick Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12, 2011.

[CCT08] G. Casale, P. Cremonesi, and R. Turrin. Robust workload estimation in queueing network performance models. In *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 183–187, Feb 2008.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *7th Symposium on*

*Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 205–218, 2006.

[CDS10] Paolo Cremonesi, Kanika Dhyani, and Andrea Sansottera. Service time estimation with a refinement enhanced hybrid clustering algorithm. In *Analytical and Stochastic Modeling Techniques and Applications*, pages 291–305, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[CG86] A. E. Conway and N. D. Georganas. Recal—a new efficient algorithm for the exact analysis of multiple-chain closed queuing networks. *J. ACM*, 33(4):768–791, August 1986.

[CGHT07] Allan Clark, Stephen Gilmore, Jane Hillston, and Mirco Tribastone. *Stochastic Process Algebras*, pages 132–179. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[CGT97] A Conn, Nick Gould, and Ph Toint. A globally convergent lagrangian barrier algorithm for optimization with general inequality constraints and simple bounds. *Mathematics of Computation of the American Mathematical Society*, 66(217):261–288, 1997.

[CHW75] K. M. Chandy, U. Herzog, and L. Woo. Parametric analysis of queuing networks. *IBM Journal of Research and Development*, 19(1):36–42, Jan 1975.

[CIAP12] Houssem-eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and Maria S. Perez. Harmony: Towards Automated Self-Adaptive Consistency in Cloud Storage. In *2012 IEEE International Conference on Cluster Computing*, volume 2012, pages 293–301. IEEE, sep 2012.

[CKL15] Artem Chebotko, Andrey Kashlev, and Shiyong Lu. A Big Data Modeling Methodology for Apache Cassandra. *2015 IEEE International Congress on Big Data*, pages 238–245, 2015.

[CKOG10] Dirceu Cavendish, Hiroshi Koide, Yuji Oie, and Mario Gerla. A mean value analysis approach to transaction performance evaluation of multi-server systems. *Concurrency and Computation: Practice and Experience*, 22(10):1267–1285, 2010.

[CL02] US Connie and GW Lloyd. *Performance solutions: a practical guide to creating responsive, scalable software*. Addison-Wesley, Reading, 2002.

[CL03] Euisun Choi and Chulhee Lee. Feature extraction based on the bhattacharyya distance. *Pattern Recognition*, 36(8):1703 – 1709, 2003.

[CLC+15] Hsueh-Hsien Chang, Meng-Chien Lee, Nanming Chen, Chao-Lin Chien, and Wei-Jen Lee. Feature extraction based hellinger distance algorithm for non-intrusive aging load identification in residential buildings. In *Industry Applications Society Annual Meeting, 2015 IEEE*, pages 1–8. IEEE, 2015.

[CM13] M. Chalkiadaki and K. Magoutis. Managing service performance in the cassandra distributed storage system. In *Proceedings of the 2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 1, pages 64–71, Dec 2013.

[CMM+13] Francisco Cruz, Francisco Maia, Miguel Matos, Rui Oliveira, João Paulo, José Pereira, and Ricardo Vilaça. Met: Workload aware elasticity for nosql. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 183–196, New York, NY, USA, 2013. ACM.

[CMRB15] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya. Workload prediction using arima model and its impact on cloud applications's qos. *IEEE Transactions on Cloud Computing*, 3(4):449–458, Oct 2015.

[CMS10] Giuliano Casale, Ningfang Mi, and Evgenia Smirni. Model-driven system capacity planning under workload burstiness. *IEEE Transactions on Computers*, 59(1):66–80, 2010.

[Coe07] *MOP Evolutionary Algorithm Approaches*, pages 61–130. Springer US, Boston, MA, 2007.

[Cou] Couchdb databases. `http://couchdb.apache.org/`.

[CRB11] R. N. Calheiros, R. Ranjan, and R. Buyya. Virtual machine provisioning based on analytical performance and qos in cloud computing environments. In *2011 International Conference on Parallel Processing*, pages 295–304, Sep. 2011.

[CS14] Paolo Cremonesi and Andrea Sansottera. Indirect estimation of service demands in the presence of structural changes. *Performance Evaluation*, 2014.

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, pages 143–154, 2010.

[CZ14] C.L. Philip Chen and Chun-Yang Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information Sciences*, 275:314 – 347, 2014.

[Dat] Datastax.

[Dat17] Datastax. Hinted handoff: repair during write path, sep 2017.

[DBC18] Salvatore Dipietro, Rajkumar Buyya, and Giuliano Casale. Pax: Partition-aware autoscaling for the cassandra nosql database. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2018.

[DCS17] Salvatore Dipietro, Giuliano Casale, and Giuseppe Serazzi. A queueing network model for performance prediction of apache cassandra. In *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools*, 2017.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Proceedings of 6th Symposium on Operating Systems Design and Implementation*, pages 137–149, 2004.

[DMGG16] Andrea De Mauro, Marco Greco, and Michele Grimaldi. A formal definition of big data based on its essential features. *Library Review*, 65(3):122–135, 2016.

[DMVRT11] Thibault Dory, Boris Mejías, Peter Van Roy, and Nam Luc Tran. Comparative elasticity and scalability measurements of cloud databases, 2011.

[DWS12] Brian Dougherty, Jules White, and Douglas C Schmidt. Model-driven auto-scaling of green cloud computing infrastructure. *Future Generation Computer Systems*, 28(2):371–378, 2012.

[ESn16] ESnet. Iperf, jun 2016.

[FLWC12] W. Fang, Z. Lu, J. Wu, and Z. Cao. Rpps: A novel resource prediction and provisioning scheme in cloud data center. In *Proceedings of the 2012 IEEE Ninth International Conference on Services Computing*, pages 609–616, June 2012.

[GGK+14] Andrea Gandini, Marco Gribaudo, William J Knottenbelt, Rasha Osman, and Pietro Piazzolla. Performance Evaluation of NoSQL Databases. In *Computer Performance Engineering*, pages 16–29. 2014.

[GH88] David E. Goldberg and John H. Holland. Genetic algorithms and machine learning. *Machine Learning*, 3(2):95–99, Oct 1988.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[Glo98] Fred Glover. A template for scatter search and path relinking. *Lecture notes in computer science*, 1363:13–54, 1998.

[GN67] William J. Gordon and Gordon F. Newell. Closed queuing systems with exponential servers. *Operations Research*, 15(2):254–265, 1967.

[Gou56] H. W. Gould. Some generalizations of vandermonde's convolution. *The American Mathematical Monthly*, 63(2):84–91, 1956.

[GPM14] Panagiotis Garefalakis, Panagiotis Papadopoulos, and Kostas Magoutis. ACaZoo: A Distributed Key-Value Store Based on Replicated LSM-Trees. In *2014 IEEE*

*33rd International Symposium on Reliable Distributed Systems*, pages 211–220. IEEE, oct 2014.

[GR11] John Gantz and David Reinsel. Extracting Value from Chaos State of the Universe: An Executive Summary. *IDC iView*, (June):1–12, 2011.

[Gra81] Jim Gray. The Transaction Concept : Virtues and Limitations. *Proceedings of the 7th International Conference on Very Large Data Bases*, (1):144–154, 1981.

[GRD04] François Goudail, Philippe Réfrégier, and Guillaume Delyon. Bhattacharyya distance as a contrast parameter for statistical processing of noisy optical images. *J. Opt. Soc. Am. A*, 21(7):1231–1240, Jul 2004.

[Har04] Peter G. Harrison. Reversed processes, product forms and a non-product form. *Linear Algebra and its Applications*, 386:359 – 381, 2004. Special Issue on the Conference on the Numerical Solution of Markov Chains 2003.

[HBa] Hbase. `http://hbase.apache.org/`.

[HC06] Sang-Jun Han and Sung-Bae Cho. Evolutionary Neural Networks for Anomaly Detection Based on the Behavior of a Program. 36(3), 2006.

[Hel09] Ernst Hellinger. Neue begründung der theorie quadratischer formen von unendlichvielen veränderlichen. *Journal für die reine und angewandte Mathematik*, 136:210–271, 1909.

[HHL12] Che-Lun Hung, Yu-Chen Hu, and Kuan-Ching Li. Auto-scaling model for cloud computing system. *International Journal of Hybrid Information Technology*, 5(2):181–186, 2012.

[HHLD11] Jing Han, E. Haihong, Guan Le, and Jian Du. Survey on NoSQL database. *Proceedings - 2011 6th International Conference on Pervasive Computing and Applications, ICPCA 2011*, pages 363–366, 2011.

[HKR13] Nikolas Roman Herbst, Samuel Kounev, and Ralf H Reussner. Elasticity in cloud computing: What it is, and what it is not. In *ICAC*, volume 13, pages 23–27, 2013.

[HL04] P. G. Harrison and Ting Ting Lee. A new recursive algorithm for computing generating functions in closed multi-class queueing networks. In *The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings.*, pages 231–238, Oct 2004.

[Hyp] Hypertable databases. `http://hypertable.org/`.

[IA15] Ishwarappa and J Anuradha. A brief introduction on big data 5Vs characteristics and hadoop technology. In *Procedia Computer Science*, volume 48, pages 319–324, 2015.

[Int16a] Intel. Intel next generation microarchitecture (nehalem), jun 2016.

[Int16b] Intel. Intel sse4 programming reference, jun 2016.

[IZE11] IBM, Paul Zikopoulos, and Chris Eaton. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill Osborne Media, 1st edition, 2011.

[JWY15] Qingchao Jiang, Bei Wang, and Xuefeng Yan. Multiblock independent component analysis integrated with hellinger distance and bayesian inference for non-gaussian plant-wide process monitoring. *Industrial & Engineering Chemistry Research*, 54(9):2497–2508, 2015.

[KAB+11] Ioannis Konstantinou, Evangelos Angelou, Christina Boumpouka, Dimitrios Tsoumakos, and Nectarios Koziris. On the elasticity of nosql databases over cloud management platforms. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 2385–2388, New York, NY, USA, 2011. ACM.

[Kai67] T. Kailath. The divergence and bhattacharyya distance measures in signal selection. *IEEE Transactions on Communication Technology*, 15(1):52–60, February 1967.

[KAT+12] Ioannis Konstantinou, Evangelos Angelou, Dimitrios Tsoumakos, Christina Boumpouka, Nectarios Koziris, and Spyros Sioutas. Tiramola: elastic nosql provisioning through a cloud management platform. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 725–728. ACM, 2012.

[KDSS16] Y. Kishore, N. H. V. Datta, K. V. Subramaniam, and D. Sitaram. Qos aware resource management for apache cassandra. In *Proceedings of the 2016 IEEE 23rd International Conference on High Performance Computing Workshops (HiPCW)*, pages 3–10, Dec 2016.

[KKR14] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. Benchmarking scalability and elasticity of distributed database systems. *Proceedings of the VLDB Endow.*, 7(12):1219–1230, August 2014.

[KKRD12] A. Kalbasi, D. Krishnamurthy, J. Rolia, and S. Dawson. Dec: Service demand estimation with confidence. *IEEE Transactions on Software Engineering*, 38(3):561–578, May 2012.

[KKRR11] A. Kalbasi, D. Krishnamurthy, J. Rolia, and M. Richter. Mode: Mix driven on-line resource demand estimation. In *2011 7th International Conference on Network and Service Management*, pages 1–9, Oct 2011.

[KM17] Flora Karniavoura and Kostas Magoutis. A measurement-based approach to performance prediction in nosql systems. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 255–262. IEEE, 2017.

[Koz10] Heiko Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634 – 658, 2010. Special Issue on Software and Performance.

[KP14] S. D. Kuznetsov and A. V. Poskonin. NoSQL data management systems. *Programming and Computer Software*, 40(6):323–332, nov 2014.

[KPSCD09] Stephan Kraft, Sergio Pacheco-Sanchez, Giuliano Casale, and Stephen Dawson. Estimating service resource consumption from response time measurements. In *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS '09, pages 48:1–48:10, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[KYTA12] A. Khan, X. Yan, S. Tao, and N. Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *2012 IEEE Network Operations and Management Symposium*, pages 1287–1294, April 2012.

[KZ06] Terence Kelly and Alex Zhang. Predicting performance in distributed enterprise applications. *HP Laboratories Technical Report HPL-2006-76*, 2006.

[KZT] Dinesh Kumar, Li Zhang, and Asser Tantawi. Enhanced inferencing: Estimation of a workload dependent performance model. VALUETOOLS '09.

[Lan01] Doug Laney. 3D Data Management: Controlling Data Volume, Velocity, and Variety. *Application Delivery Strategies*, 949(February 2001):4, 2001.

[LAV+15] João Ricardo Lourenço, Veronika Abramova, Marco Vieira, Bruno Cabral, and Jorge Bernardino. NoSQL Databases: A Software Engineering Perspective. In Alvaro Rocha, Ana Maria Correia, Sandra Costanzo, and Luis Paulo Reis, editors, *Advances in Intelligent Systems and Computing*, volume 353 of *Advances in Intelligent Systems and Computing*, pages 741–750. Springer International Publishing, Cham, 2015.

[LBMAL14] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4):559–592, Dec 2014.

[LFG05] Yan Liu, Alan Fekete, and Ian Gorton. Design-level performance prediction of component-based applications. *IEEE Trans. Softw. Eng.*, 31(11):928–941, November 2005.

[LGR15] P. Li, D. Gao, and M. K. Reiter. Replica placement for availability in the worst case. In *Proceedings of the 2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 599–608, June 2015.

[LJ12] Alexandros Labrinidis and Hosagrahar V Jagadish. Challenges and opportunities with big data. *Proceedings of the VLDB Endowment*, 5(12):2032–2033, 2012.

[LK14] Rodolfo Lourenzutti and Renato A Krohling. The hellinger distance in multicriteria decision making: An illustration to the topsis and todim methods. *Expert Systems with Applications*, 41(9):4414–4421, 2014.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[Lor16] Mark Lord. hdparm, jun 2016.

[LR80] S. S. Lavenberg and M. Reiser. Stationary state probabilities at arrival instants for closed queueing networks with multiple types of customers. *Journal of Applied Probability*, 17(4):10481061, 1980.

[LRS+14] Si Liu, Muntasir Raihan Rahman, Stephen Skeirik, Indranil Gupta, and José Meseguer. Formal modeling and analysis of cassandra in maude. In *International Conference on Formal Engineering Methods*, pages 332–347. Springer, 2014.

[LWXZ06] Zhen Liu, Laura Wynter, Cathy H. Xia, and Fan Zhang. Parameter inference of queueing models for it systems using end-to-end measurements. *Performance Evaluation*, 63(1):36 – 60, 2006.

[LXMZ03] Zhen Liu, CH Xia, P Momcilovic, and L Zhang. Ambience: Automatic model building using inference. In *Congress MSR03*, 2003.

[LZGS84] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[LZL+15] P. Lu, L. Zhang, X. Liu, J. Yao, and Z. Zhu. Highly efficient data migration and backup for big data applications in elastic optical inter-data-center networks. *IEEE Network*, 29(5):36–42, Sep. 2015.

[MADD04] Daniel A Menasce, Virgilio AF Almeida, Lawrence W Dowdy, and Larry Dowdy. *Performance by design: computer capacity planning by example*. Prentice Hall Professional, 2004.

[Man04] Jason Manning. Apache Storm, 2004.

[MAS19] Pedro Martins, Maryam Abbasi, and Filipe Sá. A study over nosql performance. In *World Conference on Information Systems and Technologies*, pages 603–611. Springer, 2019.

[MCB+11] James Manyika, Michael Chui, Jacques Bughin, Richard Dobbs, Peter Bisson, and Alex Marrs. Big data: The next frontier for innovation, competition, and productivity — McKinsey & Company. Technical Report June, 2011.

[Men08] Daniel A Menasce. Computing missing service demand parameters for performance models. In *Int. CMG Conference*, pages 241–248, 2008.

[MH13] A B M Moniruzzaman and Syed Akhter Hossain. NoSQL Database : New Era of Databases for Big data Analytics- Classification , Characteristics and Comparison. *International Journal of Database Theory and Application*, 6(4):1–13, 2013.

[Mit98] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[Mon] Mongodb databases. `https://www.mongodb.org/`.

[MPGSL06] Josefa Mula, Raul Poler, Jose P García-Sabater, and Francisco Cruz Lario. Models for production planning under uncertainty: A review. *International journal of production economics*, 103(1):271–285, 2006.

[MPV12] D.C. Montgomery, E.A. Peck, and G.G. Vining. *Introduction to Linear Regression Analysis*. Wiley Series in Probability an. Wiley, 2012.

[MRSJ14] Prashanth Menon, Tilmann Rabl, Mohammad Sadoghi, and Hans Arno Jacobsen. CaSSanDra: An SSD boosted key-value store. *Proceedings - International Conference on Data Engineering*, pages 1162–1167, 2014.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[MyS] Mysql databases. `http://www.mysql.com/`.

[MZR+07] Ningfang Mi, Qi Zhang, Alma Riska, Evgenia Smirni, and Erik Riedel. Performance impacts of autocorrelated flows in multi-tiered systems. *Performance Evaluation*, 64(9-12):1082–1101, 2007.

[neo] Neo4j databases. `http://neo4j.com`.

[Nie16] Raik Niemann. Towards the Prediction of the Performance and Energy Efficiency of Distributed Data Management Systems. In *Companion Publication for ACM/SPEC on International Conference on Performance Engineering - ICPE '16 Companion*, pages 23–28, New York, New York, USA, 2016. ACM Press.

[Nik01] Mikhail S Nikulin. Hellinger distance. *Encyclopedia of mathematics*, 78, 2001.

[NKJT09] Ramon Nou, Samuel Kounev, Ferran Juli, and Jordi Torres. Autonomic qos control in enterprise grid environments using online simulation. *Journal of Systems and Software*, 82(3):486 – 502, 2009.

[NKY+11] Y. Nakamizo, H. Koide, K. Yoshinaga, D. Cavendish, and Y. Oie. Mva modeling of multi-core server distributed systems. In *2011 Third International Conference on Intelligent Networking and Collaborative Systems*, pages 617–620, Nov 2011.

[NL18] Trong-Dat Nguyen and Sang-Won Lee. Optimizing mongodb using multi-streamed ssd. In *Proceedings of the 7th International Conference on Emerging Databases*, pages 1–13. Springer, 2018.

[NSG+15] A. Naskos, E. Stachtiari, A. Gounaris, P. Katsaros, D. Tsoumakos, I. Konstantinou, and S. Sioutas. Harmon. In *Proceedings of the 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 31–40, May 2015.

[OBS99] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley db. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[OK12] Rasha Osman and William J. Knottenbelt. Database system performance evaluation models: A survey. *Performance Evaluation*, 69(10):471–493, 2012.

[OP14] Rasha Osman and Pietro Piazzolla. Modelling Replication in NoSQL Datastores. In *Quantitative Evaluation of Systems*, pages 194–209. Springer International Publishing, 2014.

[Ora] Oracle databases. `http://www.oracle.com/us/products/database/overview/index.html`.

[PC17] J. F. Prez and G. Casale. Line: Evaluating software applications in unreliable environments. *IEEE Transactions on Reliability*, 2017.

[PCPS15] Juan F Pérez, Giuliano Casale, and Sergio Pacheco-Sanchez. Estimating computational requirements in multi-threaded applications. *IEEE Transactions on Software Engineering*, 41(3):264–278, 2015.

[PHK17] Iker Perez, David Hodge, and Theodore Kypraios. Auxiliary variables for bayesian inference in multi-class queueing networks. *Statistics and Computing*, Nov 2017.

[Pok13] Jaroslav Pokorny. Nosql databases: a step to database scalability in web environment. *International Journal of Web Information Systems*, 9(1):69–82, 2013.

[PPSC13] Juan F. Perez, Sergio Pacheco-Sanchez, and Giuliano Casale. An Offline Demand Estimation Method for Multi-threaded Applications. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 21–30. IEEE, aug 2013.

[PSST08] Giovanni Pacifici, Wolfgang Segmuller, Mike Spreitzer, and Asser Tantawi. Cpu demand for web serving: Measurement analysis and dynamic estimation. *Performance Evaluation*, 65(6):531 – 553, 2008. Innovative Performance Evaluation Methodologies and Tools: Selected Papers from ValueTools 2006.

[QCB16] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys*, 2016.

[QP15] Zhan Qiu and Juan F. Pérez. Evaluating the effectiveness of replication for tail-tolerance. In *CCGrid*, pages 443–452. IEEE Computer Society, 2015.

[Red] Redis databases. `http://http://redis.io/`.

[Rén61] Alfréd Rényi. On measures of information and entropy. In *Proceedings of the 4th Berkeley symposium on mathematics, statistics and probability*, volume 1, 1961.

[RGVS+12] Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, Victor Muntés-Mulero, Hans-Arno Jacobsen, and Serge Mankovskii. Solving big data challenges for enterprise application performance management. *Proceedings of the VLDB Endowment*, 5(12):1724–1735, August 2012.

[Ria] Riak databases. `http://basho.com/products/`.

[RK75] M. Reiser and H. Kobayashi. Queuing networks with multiple closed chains: Theory and computational algorithms. *IBM Journal of Research and Development*, 19(3):283–294, May 1975.

[RL80] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *J. ACM*, 27(2):313–322, April 1980.

[RPS09] Ajith Ranabahu, Pankesh Patel, and Amit Sheth. *Service Level Agreement in Cloud Computing*. 01 2009.

[RV95] Jerome Rolia and Vidar Vetland. Parameter estimation for performance models of distributed application systems. In *Proc. of the 1995 Conference of the Centre for Advanced Studies on Collaborative Research*, 1995.

[Sau02] Roger M Sauter. In all likelihood. *Technometrics*, 44(4):404–404, 2002.

[SBC+08] Abhishek B. Sharma, Ranjita Bhagwan, Monojit Choudhury, Leana Golubchik, Ramesh Govindan, and Geoffrey M. Voelker. Automatic request categorization in internet services. *SIGMETRICS Perform. Eval. Rev.*, 36(2):16–25, August 2008.

[SCBK15] Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. Evaluating approaches to resource demand estimation. *Performance Evaluation*, 2015.

[Sch79] P Schweitzer. Approximate analysis of multiclass closed networks of queues" presented at the. In *International Conference on Stochastic Control and Optimization*, 1979.

[Sim] Simpledb databases. `https://aws.amazon.com/simpledb/`.

[Sim06] Dan Simon. *Optimal state estimation: Kalman, H infinity, and nonlinear approaches*. John Wiley & Sons, 2006.

[SJ11] Charles Sutton and Michael I. Jordan. Bayesian inference for queueing networks and modeling of internet services. *The Annals of Applied Statistics*, 5(1):254–282, 2011.

[SKG+12] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 18–18, Berkeley, CA, USA, 2012. USENIX Association.

[SKZ07] Christopher Stewart, Terence Kelly, and Alex Zhang. Exploiting nonstationarity for performance prediction. *SIGOPS Oper. Syst. Rev.*, 41(3):31–44, March 2007.

[SM81] K. C. Sevcik and I. Mitrani. The distribution of queuing network states at input and output instants. *J. ACM*, 28(2):358–371, April 1981.

[SQL] Microsoft sql server databases. `http://www.microsoft.com/en-us/sqlserver/default.aspx`.

[SSV07] Rajan Suri, Sushanta Sahu, and Mary Vernon. Approximate mean value analysis for closed queuing networks with multiple-server stations. In *Proceedings of the 2007 Industrial Engineering Research Conference*, 2007.

[SV16] I. Sason and S. Verd. $f$ -divergence inequalities. *IEEE Transactions on Information Theory*, 62(11):5973–6006, Nov 2016.

[SWED16] D. Seybold, N. Wagner, B. Erb, and J. Domaschka. Is elasticity of scalable databases a myth? In *Proceedings of the 2016 IEEE International Conference on Big Data (Big Data)*, pages 2827–2836, Dec 2016.

[Tcp18] Tcpdump. Tcpdump, 2018.

[THIC11] Juan M Tirado, Daniel Higuero, Florin Isaila, and Jesus Carretero. Predictive data grouping and placement for cloud-based elastic server infrastructures. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 285–294. IEEE Computer Society, 2011.

[TKB+13] D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris. Automated, elastic resource provisioning for nosql clusters using tiramola. In *Proceedings of the 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 34–41, May 2013.

[TW79] Kishor S. Trivedi and Robert A. Wagner. A decision model for closed queuing networks. *IEEE Transactions on Software Engineering*, (4):328–332, 1979.

[ULP+07] Zsolt Ugray, Leon Lasdon, John Plummer, Fred Glover, James Kelly, and Rafael Mart. Scatter search and local nlp solvers: A multistart framework for global optimization. *INFORMS Journal on Computing*, 19(3):328–340, 2007.

[UPS+07] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. Analytic modeling of multitier internet applications. *ACM Trans. Web*, 1(1), May 2007.

[WC13] W. Wang and G. Casale. Bayesian service demand estimation using gibbs sampling. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 567–576, Aug 2013.

[WCKN16] Weikun Wang, Giuliano Casale, Ajay Kattepur, and Manoj Nambiar. Maximum likelihood estimation of closed queueing network demands from queue length data. In *Proc. of the 7th ACM/SPEC on International Conference on Performance Engineering*, 2016.

[WCKN18] Weikun Wang, Giuliano Casale, Ajay Kattepur, and Manoj K. Nambiar. QMLE: A methodology for statistical inference of service demands from queueing data. *TOMPECS*, 2018.

[WFZ+11] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective. *Cidr*, pages 134–143, 2011.

[WHQ+12] W. Wang, X. Huang, X. Qin, W. Zhang, J. Wei, and H. Zhong. Application-level cpu consumption estimation: Towards performance isolation of multi-tenancy web applications. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 439–446, June 2012.

[WLZZ14] Huajin Wang, Jianhui Li, Haiming Zhang, and Yuanchun Zhou. Benchmarking replication and consistency strategies in cloud serving databases: Hbase and cassandra. In *Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, pages 71–82. Springer, 2014.

[WMNO06] Richard A Waltz, José Luis Morales, Jorge Nocedal, and Dominique Orban. An interior algorithm for nonlinear optimization that combines line search and trust region steps. *Mathematical programming*, 107(3):391–408, 2006.

[Won78] J. W. Wong. Queueing network modeling of computer communication networks. *ACM Comput. Surv.*, 10(3):343–351, September 1978.

[WPC15] Weikun Wang, Juan F Pérez, and Giuliano Casale. Filling the gap: a tool to automate parameter estimation for software performance models. In *Proceedings of the 1st International Workshop on Quality-Aware DevOps*, pages 31–32. ACM, 2015.

[WZL05] Murray Woodside, Tao Zheng, and Marin Litoiu. The use of optimal filters to track parameters of performance models. In *null*, pages 74–84. IEEE, 2005.

[WZL06] M. Woodside, Tao Zheng, and M. Litoiu. Service system resource management based on a tracked layered performance model. In *2006 IEEE International Conference on Autonomic Computing*, pages 175–184, June 2006.

[YG07] Haifeng Yu and Phillip B. Gibbons. Optimal inter-object correlation when replicating for availability. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 254–263, New York, NY, USA, 2007. ACM.

[YGN06] Haifeng Yu, Phillip B Gibbons, and Suman Nath. Availability of multi-object operations. In *Proceedings of the 3 USENIX Symposium on Networked Systems Design and Implementation*, pages 211–224, 2006.

[YLL09] C. H. You, K. A. Lee, and H. Li. An svm kernel with gmm-supervector based on the bhattacharyya distance for speaker recognition. *IEEE Signal Processing Letters*, 16(1):49–52, Jan 2009.

[Zah79] John Zahorjan. An exact solution method for the general class of closed separable queueing networks. In *Proc. of Conference on Simulation, Measurement and Modeling of Computer Systems*, 1979.

[ZCDD12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, and Ankur Dave. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *Nsdi*, pages 2–2, 2012.

[ZCS07] Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Fourth International Conference on Autonomic Computing (ICAC'07)*, pages 27–27, June 2007.

[ZE11] Paul Zikopoulos and Chris Eaton. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data: Analytics for Enterprise Class Hadoop and Streaming Data*. 2011.

[ZM] P. Zarchan and H. Musoff. *Fundamentals of Kalman Filtering: A Practical Approach*.

[ZWL08] T. Zheng, C. M. Woodside, and M. Litoiu. Performance model estimation and tracking using optimal filters. *IEEE Transactions on Software Engineering*, 34(3):391–406, May 2008.

[ZYW+05] Tao Zheng, Jinmei Yang, Murray Woodside, Marin Litoiu, and Gabriel Iszlai. Tracking time-varying parameters in software systems with extended kalman filters. In *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '05, pages 334–345. IBM Press, 2005.

[ZZBH13] Qi Zhang, Mohamed Faten Zhani, Raouf Boutaba, and Joseph L Hellerstein. Harmony: Dynamic heterogeneity-aware resource provisioning in the cloud. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems (ICDCS),*, pages 510–519. IEEE, 2013.