Universidade Federal do Ceará
Centro de Ciências
Departamento de Computação
Mestrado e Doutorado em Ciência da Computação

BENCHXTEND: A TOOL TO MEASURE THE ELASTICITY OF CLOUD DATABASE SYSTEMS

Rodrigo Felix de Almeida

DISSERTAÇÃO DE MESTRADO

Fortaleza
Setembro - 2013

Universidade Federal do Ceará
Centro de Ciências
Departamento de Computação

Rodrigo Felix de Almeida

BENCHXTEND: A TOOL TO MEASURE THE ELASTICITY OF CLOUD DATABASE SYSTEMS

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal do Ceará como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Javam de Castro Machado, DSc
Co-orientadores: Prof. Flávio R. C. Sousa, DSc.

Fortaleza
Setembro - 2013

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca de Ciências e Tecnologia

A45b Almeida, Rodrigo Félix de.
BenchXtend: a tool to measure the elasticity of cloud database systems. / Rodrigo Félix de Almeida. – 2013.
87f. : il., color., enc. ; 30 cm.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Departamento de Computação, Programa de Pós Graduação em Ciência da Computação, Fortaleza, 2013.
Área de Concentração: Sistemas de informação.
Orientação: Prof. Dr. Javan de Castro Machado.
Coorientação: Prof. Dr, Flávio R. C. Sousa.
.

1. Banco de dados - Gerência. 2. Computação em nuvem. I. Título.

CDD 005

BenchXtend: a tool to measure the elasticity of cloud database systems

Rodrigo Felix de Almeida

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal do Ceará como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Prof. Javam Machado, DSc
Universidade Federal do Ceará

Prof. José Maria da Silva Monteiro, DSc
Universidade Federal do Ceará

Prof. Flávio R. C. Sousa, DSc
Universidade Federal do Ceará

Prof. Sérgio Lifschitz, DSc
Pontifícia Universidade Católica - Rio de Janeiro

Aprovada em 27 de Setembro de 2013

Aos Meus Pais.

ACKNOWLEDGEMENTS

Ao meu orientador, prof. Javam Machado, por ter confiado no meu potencial como estudante de mestrado e por ter sido compreensivo nos momentos delicados em que tive que conciliar as atividades de mestrado com o meu trabalho como gerente de projetos no LSBD.

Aos meus pais, Sebastião e Escolástica, por terem dado todo o suporte e base familiar necessários para que eu pudesse focar nas atividades, durante boa parte do meu mestrado.

À minha amada esposa, Bárbara Vasconcelos, que passou de noiva a esposa durante esse percurso da pós-graduação, por ter estado sempre comigo, compreendido os momentos de ausência ou falta de atenção e me aconselhado nos momentos em que vacilei sobre a continuação desse projeto.

Aos professores Leonardo Moreira, por acompanhar e sugerir melhorias no trabalho, e Flávio Sousa por estar sempre presente e solícito no acompanhamento e direcionamento dos detalhes da pesquisa, sendo primordial para os resultados obtidos.

A todos os colaboradores do LSBD por terem me ajudado e desempenhado com competência as atividades dos projetos nos momentos em que tive que me ausentar, devido aos compromissos do mestrado.

Ao LSBD como instituição por ter fornecido uma estrutura física adequada para a pesquisa e pelo apoio financeiro para participação de congressos científicos.

Aos colegas de mestrado da UFC por compartilharem as alegrias e as tristezas durante as disciplinas e as escritas de dissertações e de artigos.

Aos professores José Maria Monteiro e Sérgio Lifschitz por comporem a banca e pelas valiosas sugestões no aprimoramento desta dissertação.

A Deus, por estar sempre ao meu lado, dando-me coragem para enfrentar todos os obstáculos da vida.

Success is getting what you want.
Happiness is wanting what you get.
—DALE CARNEGIE

RESUMO

Nos últimos anos, a computação em nuvem tem atraído a atenção tanto da indústria quanto do meio acadêmico, tornando-se comum encontrar na literatura relatos de adoção de computação em nuvem por parte de empresas e instituições acadêmicas. Uma vez que a maioria das aplicações em nuvem são orientadas a dados, sistemas de gerenciamento de bancos de dados são componentes críticos das aplicações. Novos sistemas de bancos de dados surgiram para atender a novos requisitos de aplicações altamente escaláveis em nuvem. Esses sistemas possuem diferenças marcantes quando comparados com sistemas relacionais tradicionais. Além disso, uma vez que elasticidade é um recurso chave da computação em nuvem e um diferencial desse paradigma, esses novos sistemas de bancos de dados também devem prover elasticidade. Juntamente com o surgimento desses novos sistemas, surge também a necessidade de avaliá-los.

Ferramentas tradicionais de benchmark para bancos de dados não são suficientes para analisar as especificidades desses sistemas em nuvem. Assim, novas ferramentas de benchmark são necessárias para avaliar adequadamente esses sistemas em nuvem e como medir o quão elásticos eles são. Antes de avaliar e calcular a elasticidade desses sistemas, se faz necessária a definição de um modelo com métricas de elasticidade que façam sentido tanto para consumidores quanto provedores.

Nesse trabalho apresentamos BenchXtend, uma ferramenta, que estende o Yahoo! Cloud Serving Benchmark (YCSB), para benchmarking e medição de elasticidade de bancos de dados em nuvem. Como parte desse trabalho, propomos um modelo com métricas a partir das perspectivas dos consumidores e dos provedores para medir a elasticidade. Por fim, avaliamos nossa solução através de experimentos e verificamos que nossa ferramenta foi capaz de variar a carga de trabalho, como esperado, e que nossas métricas conseguiram capturar a variação de elasticidade nos cenários analisados.

Palavras-chave: Benchmarking, Elasticidade, Bancos de dados, Computação em Nuvem

ABSTRACT

In recent years, cloud computing has attracted attention from industry and academic world, becoming increasingly common to find cases of cloud adoption by companies and research institutions in the literature. Since the majority of cloud applications are data-driven, database management systems powering these applications are critical components in the application stack. Many novel database systems have emerged to fulfill new requirements of high-scalable cloud applications. Those systems have remarkable differences when compared to traditional relational databases. Moreover, since elasticity is a key feature in cloud computing and it is a differential of this computing paradigm, novel database systems must also provide elasticity. Altogether with the emergence of these new systems, the need of evaluating them comes up.

Traditional benchmark tools for database systems are not sufficient to analyze some specificities of these systems in a cloud. Thus, new benchmark tools are required to properly evaluate such cloud systems and also to measure how elastic they are. Before actually benchmarking and measuring elasticity of cloud database systems, it becomes necessary to define a model with elasticity metrics that makes sense both for consumers and providers.

In this work we present BenchXtend, a tool, that extends Yahoo! Cloud Serving Benchmark (YCSB), to benchmark cloud database systems and to measure elasticity of such systems. As part of this work, we propose a model with metrics from consumer and provider perspectives to measure elasticity. Finally, we evaluated our solution by performing experiments and we verified that our tool could properly vary the load during execution, as expected, and that our elasticity model could capture the elasticity differences between the studied scenarios.

Keywords: Benchmarking, Elasticity, Databases, Cloud

CONTENTS

LIST OF ABBREVIATIONS

API - Application Programming Interface

ACID - Atomicity, Consistency, Isolation and Durability

DaaS - Database-as-a-Service

DBMS - DataBase Management System

EBS - Elastic Block Store

EC2 - Elastic Compute Cloud

ECU - EC2 Compute Unit

IaaS - Infrastrucure-as-a-Service

IOPS - I/O operations per second

OLTP - On Line Transaction Processing

OLAP - On Line Analytical Processing

PaaS - Platform-as-a-Service

QoS - Quality of Service

RDBMS - Relational DataBase Management System

SaaS - Software-as-a-Service

SLA - Service Level Agreement

vCPU - Virtual Central Processing Unit

VM - Virtual Machine

XML - eXtensible Markup Language

YCSB - Yahoo! Cloud Serving Benchmark

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Scalability, pay-per-use or pay-as-you-go pricing model and elasticity are the major reasons for the successful and widespread adoption of cloud infrastructures. Since the majority of cloud applications are data-driven, database management systems (DBMSs) powering these applications are critical components in the cloud software stack [1]. Scalability is not a new requirement of database systems for cloud environments. However, the appearance of infinite computing resources available on demand in a cloud changes the dimensions of the scalability requirement. Pay-per-use model of cloud computing allows an organization to pay by the hour for computing resources, potentially leading to cost savings even if the hourly rate to rent a machine from a provider is higher than the rate to own one [2]. This model can be applied also for database services provided in the cloud [3]. The elasticity concept encompasses the idea of scale in and out resources on demand to keep the quality of a provided service. Cloud database systems should be able to transparently manage and utilize the elastic computing resources to deal with fluctuating workloads [4]. There is currently a lot of interest in elastic database systems [5].

Scalability of a system only provides a guarantee that a system can be scaled up from a few machines to a larger number of machines. In cloud computing platforms, it is necessary to support an additional property so that scalability can be dynamically provisioned without causing any interruption in the service. Elasticity emcompasses scalability aspects and goes beyond by adding the requirement of scaling down when level of demand is low. Time is also a central aspect in elasticity, which depends on the speed of response to changed workload, while scalability allows the system to meet the changed load as long as it needs. Therefore, elasticity receives remarkable importance on cloud services.

Stateful systems, such as DBMSs, are hard to scale elastically due to the requirement of maintaining consistency of the database that they manage [5]. Many novel database systems have emerged to fulfill some needs of cloud applications. These new

database systems present several differences when compared to traditional relational systems, regarding to their data models, consistency, availability, replication strategy and so on. Since there are many differences and many available database systems, having a way to compare them is very useful for developers and architects of cloud applications, for instance. Benchmark tools are commonly used to evaluate the performance of a system as well as to help on tuning it.

The most prominent examples to evaluate transactional database systems are the various TPC benchmarks. However, TPC-family benchmarks do not consider essential aspects of cloud database systems, like the variation of demand and resources during a workload and the measurement of specific characteristics like elasticity. Defining a workload that changes the number of clients during workload process illustrates a more realistic scenario for many applications, like Web applications, in which this number goes up and down continuously. Particularly, reducing the number of clients is a more challenging aspect, that is not addressed for most related works. Therefore, to properly evaluate a cloud database system it becomes necessary to have a benchmark tool that properly fulfills cloud requirements. Before measuring elasticity of cloud database systems, a model with metrics is required. Thus, there must be defined a quantitative model to compare elasticity of different database systems. Services on cloud computing, including database services but not limited to, are sold by providers and contracted by consumers. Due to the large number of providers and to the wide range of possible consumer applications, comparing options based on some criteria help on taking decisions. On one hand, consumers want to compare cloud database services to choose one that fits better to their needs. On the other hand, providers want to meet the Service-Level Agreement (SLA) established with the consumer, regarding to the database systems, with the minimum cost and amount of resources. Therefore, the elasticity metrics must make sense both for consumer and provider perspectives in order to support them on their decisions.

In this dissertation we present a benchmark tool named BenchXtend [6] to evaluate cloud database systems and to calculate elasticity metrics of them. In order to calculate elasticity we also present a model for measuring elasticity of cloud database systems, from two perspectives: the consumer one and the provider one. This tool is an extension of YCSB [7] benchmark and aims to (i) calculate metrics to measure the elasticity of relational or non-relational database systems, based on expected quality of service (QoS) agreed in an SLA, and to (ii) provide more realistic workloads allowing to change the number of clients during a workload execution.

1.1 MOTIVATION

For decades, relational database management systems (RDBMSs) have been considered as the one-size-fits-all solution for providing data persistence and retrieval. However, the ever increasing need for scalability and new application requirements have created new challenges for traditional RDBMSs. NoSQL database systems have emerged as an alternative to reach high scalability for applications that may relax some requirements, like strong consistency. Dozens of NoSQL systems are currently available and several differences can be pointed out among the existing alternatives. Some NoSQL systems differ on their data models that could be, for instance, key-value stores, column stores or document stores. Some systems are optimized to write operations while other ones for read operations. Synchronous replication can be adopted for some of them, while other systems adopt assynchronous replication. Thus, many characteristics, that are not covered by traditional benchmark tools, should be considered by new solutions to evaluate performance of such novel systems.

YCSB [7] is an outstanding solution to benchmark NoSQL systems. YCSB has been referenced by many works and currently allows to benchmark more than ten different database systems. However, YCSB presents some limitations regarding to on how to properly emulate the behavior of applications in a cloud. For instance, YCSB does not allow to vary the load during a workload execution. If the load is not changed during an experiment, it is very hard to evaluate how a system would react to adapt itself to maintain Quality of Service (QoS) when there are some fluctuations of demand. In a cloud environment, maintenance of QoS is usually related to satisfying a contract, namely Service-Level Agreement (SLA), that establishes what are the criteria and the thresholds to state whether the quality of a service was violated or not. Therefore, adding features on YCSB or in other benchmark tool to provide a way to vary load during a workload may help one to evaluate NoSQL systems in a cloud.

Elasticity has been widely advertised by cloud providers but they do not provide a measurement of how elastic they are. If there is no metric to say how elastic a service is, one could state only that a service is elastic or inelastic. From the moment in which many cloud providers are said to provide elastic services, there should have a way to measure their actual elasticity in order to compare them. Since database systems can be provided as a cloud service and databases are usually present in most of cloud applications, measuring the elasticity of database systems becomes relevant. Therefore, a model that

represents and measures the elasticity of cloud database systems can help on identifying which services are more elastic. If a provider maintains its QoS by satisfying the SLA and tries to minimize the resource usage in a scenario with remarkable load variation, we can say the system acts in an elastic way. We argue that an elasticity model can be based on how much the SLA is met. Besides having an elasticity model, it becomes necessary to have also a benchmark tool that can run some workloads and then measure the elasticity based on the proposed model.

1.2 CONTRIBUTIONS

The major contributions of this master dissertation are:

1. Development of a benchmark tool for cloud database system that calculates elasticity metrics and varies upward and downward the number of active clients while executing a workload.
2. Definition of a model with metrics to measure the elasticity of database systems in cloud, from consumer and provider perspectives.
3. Evaluation of our model through some experiments, that can help on understanding some characteristics of novel data systems in a cloud.

1.3 PUBLICATIONS

During the graduate course, we have published the following papers related to the theme of this dissertation:

- [Almeida, 2012] Rodrigo Felix de Almeida. 2012. **BenchXtend: a tool to benchmark and measure elasticity of cloud databases** in 27th Simpósio Brasileiro de Bancos de Dados - SBBD 2012 - Workshop de Teses e Dissertações. São Paulo, SP, Brazil.
- [Almeida et al., 2013] Rodrigo Almeida, Flávio Sousa, Sérgio Lifschitz and Javam Machado. 2013. **On defining metrics for elasticity of cloud databases** in 28th Simpósio Brasileiro de Bancos de Dados - SBBD 2013. Recife, PE, Brazil.

1.4 ORGANIZATION

The remaining chapters are organized as follow:

- Chapter 2: presents basic concepts about cloud computing, relational and NoSQL databases, as well as benchmark tools and finally presents related work.
- Chapter 3: describes the proposed model of metrics for elasticity of databases in cloud, from consumer and provider perspectives.
- Chapter 4: presents the BenchXtend tool, explaining its architecture and extensions developed.
- Chapter 5: presents the enviroment where experiments were performed and analyzes the results gathered.
- Chapter 6: summarizes the conclusions and proposes future works.

CHAPTER 2

CLOUD COMPUTING AND DATABASES

2.1 CLOUD COMPUTING

In recent years, cloud computing has attracted attention from industry and academic worlds, becoming increasingly common to find in the literature cases of cloud adoption by companies and research institutions. One of the reasons is the possibility of acquiring resources in a dynamic and elastic way. In fact, elasticity is a key feature in the cloud computing context, and perhaps what distinguishes this computing paradigm from the other ones [8]. Even though elasticity is often associated with scalability, they are different concepts and should never be used interchangeably.

A system whose performance improves proportionally to the capacity added, is said to be a scalable system. System can scale in two ways: vertical or horizontal. Vertical scale or scale-up is related to keeping unchanged the number of machines but enhancing their performances by adding CPUs, memory, disks or network bandwidth. Horizontal scale or scale-out refers to adding more machines in the cluster to increase the processing power. Vertical scale may be an unfeasible strategy due to the autonomic-management requirement of cloud infrastructures. Thus, horizontal scale has been adopted for most companies and academy works. Scalability is a static property of the system that specifies its behavior on static configuration, while elasticity is a dynamic property that allows the systems scale to be increased or released on-demand while the system remains operational.

Many authors and entities have presented definitions for elasticity. NIST [9], for instance, states that "capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand". Cooper [7] states that an elastic "system can add more capacity to a running system by deploying new instances of each component, and shifting load to them.". Sorrosal [10] defines as "Capacity at runtime by adding and removing resources without service interruption in order to handle the workload variation.". Even though all definitions mentioned are reasonable, we adopted the following definition proposed by [11]:

> *Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible.*

This definition encompasses important characteristics. Firstly, it mentions "the degree" which suggests it should be measured. Secondly, it mentions the need of adaption when the load varies and then that resources can be allocated or deallocated. Besides, the adaption must be performed in an autonomic way. Finally, it states that resources match the demand being as closely as possible, i.e. allocating when needed, but trying to save resources when possible.

In cloud systems, where resources are usually managed in an autonomous way, the control and the actions related to resource management are taken in such a way to satisfy the SLA. On one hand, if SLA is not met, the system may have provisioned less resources than necessary and, consequently, the system component responsible for adding resources is not acting according to the demand. If the SLA is not being satisfied, the system mechanism responsible for monitoring and taking decisions is not well tuned to act and add machines when necessary. Therefore, the system is not being elastic and we can establish a correspondence between satisfying the SLA and system elasticity. On the other hand, if that component is providing more resources than needed, SLA is being satisfied, but the provider may be spending more resources and, consequently more money, than necessary. In this scenario resources are not as closely as possible to the demand, implying the system is not being elastic. If the provider defines a lower bound of such a metric, like number of machines, CPU usage, query response time or any other, it can help it on determining how much resource is being wasted. This bound may be defined in the SLA, but it would not generate any penalty neither for the provider nor for the consumer, if the measurements are below the lower bound.

Elasticity is also related to the speed of adding or removing resources when necessary [12]. Thus, if a system takes a long time to add resources, we can say that its elasticity is problematic or it should be improved. In this case, when the system is under-provisioned and it takes a long time to react and to stabilize itself, the system will stay more time under a state that is likely not satisfying the SLA. It also corroborates the relation between elasticity and SLA satisfaction.

2.2 DATABASES ON CLOUD

Since most cloud applications are data-driven, database management systems (DBMSs) powering these applications are critical components in the cloud software stack [1]. Data-intensive applications can be classified, in general, into two groups: Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP). OLTP systems are characterized by a large number of short transactions that can update or retrieve data. Since in OLTP systems, the database is usually accessed by concurrent users that can be both updating or reading data, to maintain the data consistency of these kinds systems is a critical factor to be treated. OLAP systems aim to work with data consolidation and data analysis, typically relaxing normalization of the modeling, performing much more read than update operations and they are usually accessed by a very few number of users. As far as cloud is concerned, some works focus on OLTP databases [7] [4] in cloud platforms, while other present analysis of OLAP databases [13] [14] in such platforms.

Stateful systems, such as DBMSs, are hard to scale elastically because of the requirement of maintaining consistency of the database that they manage [5]. However, cloud database systems must provide elasticity otherwise data-driven applications would miss some benefits that cloud computing can provide to them. Even though the adopted definition of elasticity is not focused on database-system elasticity, it can be applied if we assume that "resources" are data nodes and "workload changes" are changes on the number of queries sent to a database system.

2.2.1 Service-Level Agreements (SLAs)

Many companies expect cloud database providers to guarantee quality of service using SLAs [15]. Cloud computing contracts agreed between customers and providers are usually adherent to an SLA. SLA defines the terms that must be satisfied by the provider and it works as a guarantee for the consumer. SLAs are usually based on one or more metrics that can be understood both for consumer and provider. In general, cloud providers base their SLAs only on the availability of services. Amazon EC2, for instance, has an SLA based on availability, in which it is guaranteed an uptime of at least 99.95%. If this SLA is not satisfied, Amazon generates a service credit of 10% or 30%, depending on if the uptime was between 99.95% and 99.00% or if it was less than 99.0%, respectively.

There are some models proposed for SLAs and quality of database service that

deal with data management aspects, such as [16] and [17]. However, as far as we know, there are no providers of public clouds that base their SLAs on response times or elasticity metrics of databases, for instance.

2.2.2 Relational Databases and Distributed Databases

The reason for the proliferation of DBMSs in the cloud computing space is due to the success DBMSs, particularly relational systems, have had in modeling a wide variety of applications. The key ingredients to this success are the many features DBMSs offer: overall functionality (modeling diverse types of application using the relational model which is intuitive and relatively simple), consistency (dealing with concurrent workloads without worrying about data becoming out-of-sync), performance (both high-throughput, low-latency and more than 25 years of engineering), and reliability (ensuring safety and persistence of data in the presence of different types of failures) [18]. However, in spite of RDBMS success, such systems are not easy to scale due to the requirement of maintaining consistency of the database that they manage [5]. Moreover, traditional database management systems are designed, in general, for statically provisioned infrastructures.

This difficult of scaling RDBMS is probably the main reason that motivated the development of other systems which do not provide everything that relational systems provide, but that are easier to scale. Since scalability can be the most important requirement for many applications, such applications may give up some features of relational systems in order to have a better performance. For instance, ensuring atomicity and consistency of data entities may be a responsibility of applications to make data systems more scalable. Even though there is a remarkable number of works guiding research on cloud to alternatives for RDBMS, one can say that changing some strategies can make RDBMS as elastic as NoSQL systems [19].

Database systems, regardless whether they are configured in a cloud or not, support one concept of a transaction, which guarantees that the execution's result of multiple concurrent programs leaves the database in the same state as some serial execution of the same transactions. The term ACID denotes that a transaction is atomic in that the system executes it completely or not at all; consistent in that the database remains unchanged; isolated in that the effects of incomplete execution are not exposed; and durable in that results from completed transactions survive failures [20]. In a distributed database, if a transaction modifies objects stored at multiple servers, it must obtain and

hold locks across those servers. While this is costly even if the servers are collocated, it is more costly if the servers are in different datacenters. When data is replicated, everything becomes even more complex because it is necessary to ensure that the surviving nodes in a failure scenario can determine the actions of both completed and incomplete transactions. In addition to the added costs incurred during normal execution, these measures can force a block during failures that involve network partitions, compromising availability, as the CAP theorem [21] describes.

The notions of consistency proposed in the distributed systems literature focus on a single object and are client-centric definitions. Strong consistency means that once a write request returns successfully to the client, all subsequent reads of the object, by any client, see the effect of the write, regardless of replication, failures, partitions, and so on. The term weak consistency describes any alternative that does not guarantee strong consistency for changes to individual objects. An example of weak consistency is eventual consistency. For eventual consistency, the guarantee offered is that every update is eventually applied to all copies, but there is no guarantee regarding to in which order the updates will be applied and when they will be applied.

2.2.3 NoSQL Databases

For decades, RDBMSs have been considered as the one-size-fits-all solution for providing data persistence and retrieval. However, the ever increasing need for scalability and new application requirements have created new challenges for traditional RDBMSs. Therefore, recently, a new generation of low-cost, high-performance database software that challenges the dominance of relational database management systems has emerged. These novel systems are commonly called NoSQL, for Not only SQL, systems.

As opposed to ACID transactions of RDBMS, NoSQL DBMSs follow the CAP theorem and thus their transactions conform to the BASE (Basically, Available, Soft state, Eventually consistent) principle [22]. According to the CAP theorem, a distributed database system can only choose at most two out of three properties: Consistency, Availability and tolerance to Partitions. Therefore, most of ACID transaction systems decide to compromise the strict consistency requirement. In particular, they apply a relaxed consistency policy called eventual consistency.

Two NoSQL systems, BigTable [23] powered by Google and Dynamo [3] powered

by Amazon, have inspired the development of other novel systems, like MongoDB [24], HBase [25] and Cassandra [26]. We present more details about Cassandra in this chapter, since we used it in our experiments.

Although some novel database systems are said to be elastic they do not provide mechanisms to automatically monitor the environment and take decisions on adding or removing resources, based on one or more metrics monitored, like CPU usage, memory usage, throughput, response time, and so on. Therefore, we have to add other components apart from a database system to have a more complete system that we call cloud database system, as illustrated on Figure 2.1. This system is composed of (i) an Instance Manager, (ii) a Database Manager and (iii) a pool of instances (virtual machines) that are running or available to be started. The Instance Manager, composed of a Monitor and a Decision Taker, is responsible for monitoring the pool gathering statistics, as well as for taking decision on starting or on stopping instances of the pool. The Database Manager is the access point to where queries are sent. Depending on the DBMS under test, there is no central node to receive queries and distribute. In such systems, queries are sent all over the pool and the Database Manager is a regular instance. The pool of instances is only a set of pre-configured instances that can be started or stopped by the Instance Manager. From now on, every time we mention we are benchmarking a database system, we are refering to this cloud database system just defined.

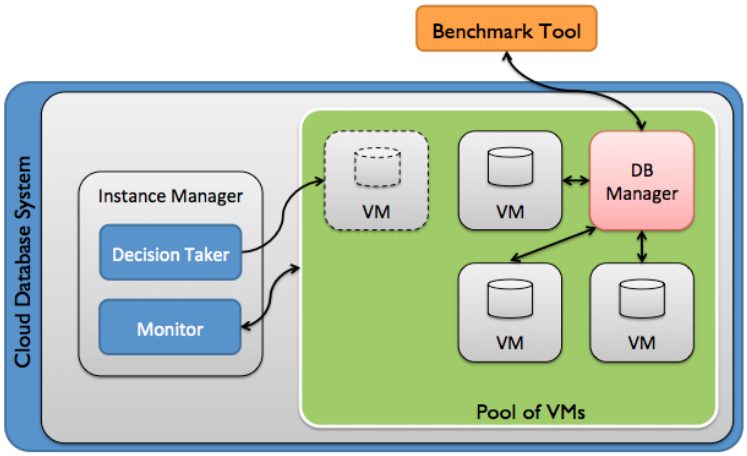

Figure 2.1: Cloud Database System

Cassandra

Apache Cassandra is an open-source, highly scalable, column-oriented, distributed database system for managing large amounts of data. Unlike relational systems, Cassan-

dra does not demand you to model all of the columns required by your application up front, as each row is not required to have the same set of columns. A Cassandra instance is a collection of independent nodes that are configured together into a cluster, where all nodes are peers, meaning there is no master node or centralized management process. A node joins a Cassandra cluster based on its configuration. This section explains key aspects of the Cassandra cluster architecture.

Cassandra uses a protocol called gossip to discover location and state information about the other nodes participating in a Cassandra cluster [27]. The gossip process runs on every second and exchanges state messages with up to three other nodes in the cluster. The nodes exchange information about themselves and about the other nodes that they have gossiped about, so all nodes quickly learn about all other nodes in the cluster. When a node first starts up, it looks at its configuration file to determine the name of the Cassandra cluster it belongs to and which nodes, called seeds, to contact to obtain information about the other nodes in the cluster. Each node owns a data range. To know what range of data it is responsible for, a node must also know its own token, that is a hash, and those of the other nodes in the cluster.

In Cassandra, the total amount of data managed by the cluster is represented as a ring, with range from 0 to $2^{217} - 1$, assuming the default partitioner, i.e. RandomPartitioner. A partitioner is a hash function for computing the token of a row key. Each row of data is uniquely identified by a row key and distributed across the cluster by the value of the token. In versions 1.1.x, the ring is divided into contiguous ranges equal to the number of nodes, as illustrated on Figure 2.2 and on Table 2.1 for a 3-node cluster. In this example, the first node, whose token is 0, has its range from $\frac{2}{3} * 2^{217}) + 1$ to 0. The token value is always equal to the end hash value of the range. The start hash value is always the value of the end hash value for the subsequent range added by 1. For instance, the start hash value of node 2 is the end hash value of node 1 plus 1, i.e. 0 + 1 = 1, as shown on Table 2.1 and graphically represented on Figure 2.2. For versions 1.2.x it is possible to use virtual nodes, or simply vnodes, feature that creates multiple ranges (256, by default, but configurable via *num_tokens* property on cassandra.yml) and evenly divides data through the nodes. This feature avoids the need of *move* operations to redefine tokens of existing nodes, which is required when a node joins or leaves the cluster. In addition, vnodes minimizes the time to recover the cluster when a node dies completely. Virtual nodes also present other advantages [28] [29].

A read or write request from a client can go to any node in the cluster. When

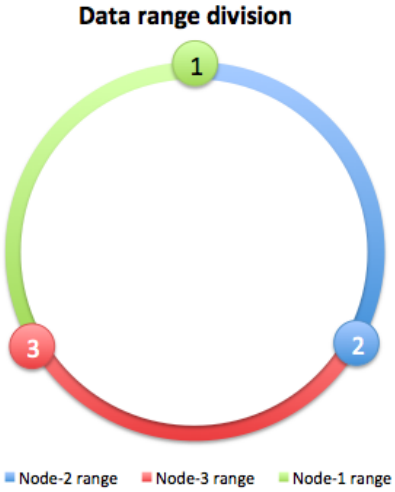

Figure 2.2: Data range division for a 3-node cluster

Node	**Token**	**Start hash value**	**End hash value**
1	0	$(\frac{2}{3} * 2^{217}) + 1$	0
2	$\frac{1}{3} * 2^{217}$	1	$(\frac{1}{3} * 2^{217})$
3	$\frac{2}{3} * 2^{217}$	$(\frac{1}{3} * 2^{217}) + 1$	$(\frac{2}{3} * 2^{217})$

Table 2.1: Tokens and range of hash values for a 3-node cluster

a client connects to a node and issues a read or write request, that node serves as the coordinator for that particular client operation. The job of the coordinator is to act as a proxy between the client application and the nodes that own the data being requested. The coordinator determines which nodes in the ring should get the request based on the cluster configured partitioner and replica placement strategy.

In Cassandra, consistency refers to how up-to-date and synchronized a row of data is on all of its replicas. Cassandra extends the concept of eventual consistency by offering tunable consistency. For any given read or write operation, the client application decides how consistent the requested data should be. For write requests, the consistency level specifies on how many replicas the write must succeed before returning an acknowledgement to the client application. For read request, the consistency level specifies how many replicas must respond before a result is returned to the client application.

Cassandra is write-optimized. Its writes are first written to a commit log for durability issues and then to an in-memory table structure called a memtable. A write is

successful once data is written to the commit log and memory, so there is very minimal disk I/O at the time of write. Writes are batched in memory and periodically written to disk to a persistent table structure called an SSTable (sorted string table). SSTables files are not changed after they are written. Thus, a row is typically stored across multiple SSTable files. This strategy has a drawback for read requests, since a row may be combined from more than one SSTable, but an in-memory structure called Bloom filter is aimed to optimized these types of requests. Periodically, Cassandra merges SSTables into larger SSTables in a process called compaction. On one hand, compaction impacts negatively on read requests since during this process there is a temporary spike on disk space usage and disk I/O. On the other hand, after the process is finished, read requests performance is improved since there are less SSTables to be checked before completing a read request.

According to our adopted definition of elasticity, Cassandra cannot be considered an elastic database, since it it not able to adapt itself depending on the workload fluctuation. However, if we add a mechanism to monitor Cassandra instances and to take decisions to add or remove instances, Cassandra can work in an autonomic way to distribute data among the nodes and to eventually maintain the consistency. Therefore, Cassandra can be a good alternative for cloud applications that need scalability and high availability without compromising performance. To the best of our knowledge, there is no all-in-one database system that is able to manage its own instances, by monitoring them and by taking decisions of changing the resources according to the load variation.

2.3 BENCHMARK TOOLS

Traditionally, the goal of benchmarking a software system is to evaluate its performance under a particular workload for a fixed configuration and to help on tuning a system. The most prominent examples for evaluating transactional database systems as well as other components on top, such as a application-servers or web-servers, are the various TPC benchmarks. All TPC-family benchmarks test environments with fixed configuration. In addition, TPC-benchmarks focus on transactional database systems that provide ACID properties.

Cloud systems have an important characteristic that is their capacity to adapt themselves depending on the variation of demand. Besides, as aforementioned, most novel data systems commonly used in a cloud do not provide ACID properties, have dif-

ferent data models, sacrifice strong consistency for availability and offer only some weaker forms of consistency. Thus, benchmark tools for cloud database systems should consider specificities of cloud computing and of new database systems available. Particularly, providing a way to vary the load to stress the system and then force it to add resources must receive attention. With such feature to vary load, benchmark tools can start thinking on how to measure elasticity. Unlike measuring efficiency, response time or throughput, to measure elasticity it is important to keep in mind that workloads must be carefully selected so that the system can be stressed. Comparing two cloud data systems with the same workload is fair only if two systems are equally or approximately stressed by the selected workload [11].

SLA plays an important role on cloud computing, since it establishes the rules between consumer and provider to guarantee quality of service. Although its importance is accepted by most works, benchmark tools usually do not consider SLA to state whether a cloud database system satisfies or not the SLA. As aforementioned, a relation can be established between poor elasticity and to dissatisfy an SLA. Therefore, benchmark tools should also match workload results with how much they are meeting the SLA.

Some tools have tried to address some specificities of benchmarking cloud database systems. The most referenced is YCSB [7], but OLTP-bench [30] also presents interesting features, although it is not properly a single benchmark.

2.3.1 YCSB

YCSB [7] framework consists of a workload generating client and a package of preconfigured Core workloads that cover interesting parts of the performance space, such as read-heavy workloads, write-heavy workloads and scan workloads. An important aspect of YCSB framework is its extensibility: the workload generator makes it easy to define new workload types, and it is also straightforward to adapt the client to benchmark new data serving systems.

YCSB provides a set of Core Workloads, as shown on Table 2.2, that model different application demands and that can be used by configuring few parameters. Each Core Workload is composed of a set of parameters like number of operations, number of threads and percentage of each operation type. Operation type percentages define the portion of queries for each operation that are expected to be performed. The available

operation types are: read, insert, update, delete, scan.

Core workload A, for instance, defines that the queries performed during workload must be evenly balanced between read and update operations, i.e. 50% for each. Such workload characterizes an update-heavy application, since in most applications the number of reads is higher than the number of writes. An example of heavy-update application is a Web site that records what a user does during his session. In this application, on one hand, the system updates frequently the recent actions of a user, but on the other hand, these actions usually require reading records from a database.

Workload	**Operations**	**Application example**
A - Update heavy	Read: 50%; Update: 50%	Session store recording recent actions in a user session
B - Read heavy	Read: 95%; Update: 5%	Photo tagging; add a tag is an update, but most operations are to read tags
C - Read only	Read: 100%	User profile cache, where profiles are constructed elsewhere (e.g., Hadoop)
D - Read latest	Read: 95%; Insert: 5%	User status updates; people want to read the latest statuses
E - Short ranges	Scan: 95%; Insert: 5%	Threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id)

Table 2.2: Core workloads that come with YCSB by default

YCSB architecture is designed in such a way it provides an abstraction layer for adapting to the API of a specific table store. To add support for new database systems few methods must be implemented, creating what is called a DB Binding. At the moment this work was written, there were fourteen DB Bindings, including bindings for well-known systems like Cassandra, HBase [25] and MongoDB [24].

While executing a workload, YCSB gathers performance metrics in order to provide, by the end of the execution, statistics about the results. YCSB also allows to choose between time series or histogram representation of the results, reporting also 95^{th} and 99^{th} percentiles, average, maximum and minimum response times by operation type. Individual query response times are not reported by the end of a workload.

2.3.2 OLTP-bench

OLTP-Bench project is a "batteries-included" benchmarking infrastructure designed for and tested on several relational DBMSs and cloud-based database-as-a-service (DBaaS) offerings. OLTP-Bench is capable of controlling transaction rate, mixture, and workload skew dynamically during the execution of an experiment, thus allowing the user to simulate a multitude of practical scenarios that are typically hard to test, for example, time-evolving access skew. Moreover, the infrastructure provides an easy way to monitor performance and resource consumption of the database system under test.

OLTP-Bench includes a set of known benchmarks, like TPC-C, Wikipedia, ResourceStresser and YCSB, in a standardized way to provide a similar configuration input file for each one and to gather comparable results.

2.4 RELATED WORK

Benchmarking databases systems is always related to performing experiments and gathering metrics. Measuring performance is one of the main purposes of benchmarking database systems. Usually, performance of such systems is measured by query response time or throughput. Considering these metrics, systems that respond to query request in less time perform better. Systems that execute more operations by second, i.e. have a higher throughput, perform better. Besides, it can be measured also the scalability of system in order to identify if the system scales linearly, for instance, as the number of resources increases linearly. These metrics are important and will be kept useful while database systems last. However, these metrics are not enough to measure the elastic characteristic of cloud database systems. Therefore, it becomes necessary to define models to represent elasticity of systems and metrics to measure it.

2.4.1 Elasticity Metrics

A number of authors has discussed about elasticity of database systems, but most of their works miss simple metrics of elasticity. [31] discusses about elastic scalability but does not present a metric for that and mention some metrics that do not consider SLA rules. [31] does not make clear what is the definition of elastic scalability and whether it

is different or not of elasticity. [32] provides metrics inspired on elasticity definition from physics, but focus on network bandwidths instead of database-specific aspects and does not use real data in their experiments.

Some authors [33] [34] discuss elasticity but do not propose metrics to compare their results with other works. [34] presents improvements when compared to [35] but the presented analysis only compare the results plotted in charts. No common indicator is used to analytically measure the results and to compare elasticity of cloud database systems. [12] proposes ways to quantify the elasticity concept in a cloud. They define a measure that reflects the financial penalty to be paid to a consumer, due to under-provisioning, by leading to unacceptable latency or throughput, or due to over-provisioning, by paying more than necessary for the resources. Nevertheless, it does not take into account DBMSs features such as query response time or throughput to calculate the metrics. It uses only a resource-oriented approach to calculate the metrics. [33] provides definitions of elasticity for database systems and a methodology to evaluate the elasticity. However, these definitions deal only with under-provisioning scenarios and do not address issues of SLA, penalties, and resources. In addition, the authors do not explain clearly how they calculate the metrics presenting arguments to support their decisions.

[36] presents a cloud-enabled framework for adaptive monitoring of NoSQL systems and it performs some experiments with YCSB for few NoSQL systems trying to vary the load by manually adding new YCSB instances. However, they do not provide metrics for elasticity, do not deal with situations of where there are more resources than necessary and present a cumbersome way to vary the load during workload execution. [7] presents the metrics named *elastic speedup* and *scaleup*. The first metric illustrates the latency variation as new machines are instantiated. The second one is a traditional metric and does not encompass elasticity aspects. Even though these metrics can be useful, they do not illustrate the consumer perspective and do not consider the variation of clients accessing an application.

2.4.2 Benchmarking Cloud Database Systems

Probably the most relevant work of benchmarking cloud database systems was proposed by Cooper [7]. YCSB is an extensible benchmark tool and has been used to compare performance of NoSQL systems in many works [37] [34] [36]. However, YCSB does not

provide a way to vary number of clients during workload execution. Thus, we believe that, without changing the workload, is not possible to measure how elastic a system is, since workload is nearly constant from the beginning to the end of the YCSB benchmark process.

[38] extends YCSB to support complex features such as including multi-tester coordination for increased load and eventual consistency measurement, multi-phase workloads to quantify the consequences of work deferment and the benefits of anticipatory configuration optimization such as B-tree pre-splitting or bulk loading. However, [38] does not provide a workload variation to stress and relax a system to measure its elasticity.

OLTP-Bench [30] presents an improvement on elasticity measurement since it allows to vary the load by varying operations/second by client. However, OLTP-Bench number of clients is fixed, what does not represent a more realistic scenario. In addition, OLTP-Bench does not calculate any elasticity metric.

In [34] the variation of load is made by removing one of the workloads being executed. However, this does not represent a regular behavior of a web application, for instance, since it is not very common to have many clients leaving an application at the same time. [39] proposes a framework that intercepts queries from application and then forward them to database layer, gathering information about the query executions. This kind of additional layer may include overheads that are hard to be measured, since no query is directly sent to the database layer. In addition, experiments show only addition of resources when a threshold is reached. Removing resources is not illustrated and analyzed. [11] clearly presents the difference of measuring elasticity, scalability and efficiency. In addition, [11] presents elasticity metrics for cloud computing and what a benchmarking for elasticity should consider. However, the elasticity metrics are focused only on the resources allocated and the expected resources, they are not aimed for database systems, and, finally, the development of the benchmark tool itself is not part of the work and details about the implementation are not presented.

Much work has been done to provide benchmark tools for cloud database systems and to propose metrics to measure elasticity. However, as we can notice there are many challenges and open issues that can be addressed by new research works. We aim to address some of these open issues in the next couple of chapters.

CHAPTER 3

ELASTICITY METRICS

According to the definition adopted in this work, elasticity is *"the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible"*. Thus, it is essential to have a model to measure the mentioned degree. Since we could not find a model with metrics for elasticity that encompasses the guarantee of quality of service and that considers consumer and provider perspectives, we propose a model with a set of metrics before actually measuring elasticity of cloud data systems with BenchXtend.

Our approach to define metrics for elasticity uses a penalty model approach to measure imperfections in elasticity for database systems. Similarly to [12], our elasticity model is composed of two parts: penalty for over- and under-provisioning. Unlike [12], we explore database system features, like query response times, and present both the consumer and provider perspectives. [12] presents the importance of analyzing the consumer point of view. [39] comments the dhicotomy of consumer and provider perspectives.

We consider in this work a scenario where a consumer accesses an available service in a Database-as-a-Service (DBaas or DaaS) provider. DBaaS is a technology where a third-party service provider hosts a database as a service [4]. Such services alleviate the need for their users to purchase expensive hardware and software, deal with software upgrades and hire professionals for administrative and maintenance tasks. From a DBaaS-consumer perspective, the database service would seamlessly scale and it would be maintained, upgraded, backed-up and handle server failure, all without impacting the consumer in any way [40]. In order to provide a complete DBaaS solution across large numbers of customers, the cloud providers need a high-degree of automation. Functions that have a regular time-based interval, like backups, can be scheduled and batched. Many other functions, such as elastic scale-out can be automated based on certain business rules. For example, providing a certain quality of service (QoS) according to the SLA might require limiting databases to a certain number of connections or a peak level

of CPU utilization, or some other criteria. When a criterion is exceeded, the DBaaS might automatically add a new database instance to share the load. DBaaS may be seen as a specific kind of service delivered by a Platform-as-a-Service (PaaS) provider [41]. [42] suggests a solution for a DBaaS. Many players like Amazon, Oracle and Microsoft already provide solutions for DBaaS. SLA receives relevant importance in this case, since it must be defined in such a way the consumer has a suitable and understandable metric to evaluate the service quality, like query response time or throughput. [39] presents some challenges on defining SLAs properly.

We do not consider a Software-as-a-service (SaaS) scenario, because, in general, a SaaS consumer has no access or control to the database system when contracts a SaaS provider. Infrastructure-as-a-service (IaaS) scenario is not considered either because the SLA is usually based only on infrastructure resources, removing from the provider the responsibility of guaranteeing quality of any software service. Amazon EC2 is an example of IaaS provider and its SLA guarantees only availability.

3.1 CONSUMER PERSPECTIVE

Due to the large number of DBaaS providers and to the so-claimed buzzword elasticity, it is important for consumers to have a model to evaluate and compare elasticity of database systems. From a DBaaS-consumer perspective, a database system is elastic if, regardless the number of queries submitted to the system, the system makes adaptions on its resources, based on the demand, in order to satisfy the SLA. Even though most providers do not give guarantees of performance on database systems yet, defining an SLA based on response time for queries can attract new consumers, since an agreement based on that allows the consumers to assure the quality of a service for the end users of their applications.

Our proposal assumes an SLA based on an agreement of response times of queries. Upper and lower bounds for response times are defined by *operation type* in the SLA. In our case, following the YCSB operations, we have the following five operation types: read, insert, update, delete, scan. For each operation type, consumers and providers can define different values, depending on what is feasible for the provider to ensure or on what the database system is optimized or not to. It is important to notice that according to this approach there should not be defined penalties related to the underlying infrastructure, like number of replicas or dedicated memory. Techniques adopted by

DBMSs, like replication or cache strategy are considered to be transparent for a DBaaS consumer. Thus, we propose metrics that abstract techniques like those. Even though we present the following metrics by operation type, the concept presented here can be applied to response times of transactions, specific queries or group of queries.

[7] presents a metric named *elastic speedup* to measure the impact on performance as new servers are manually added, while the workload is running. They stated that a system that offers good elasticity should show a performance improvement when new servers are added, with a short or non-existent period of disruption while the system is reconfiguring itself to use the new server. Even though this metric can be useful from the provider perspective, from the consumer perspective it would be more important to have a metric that considers the increase and decrease of numbers of clients, instead of number of machines. As aforementioned, for a DBaaS consumer, it is transparent how many machines are needed to meet a response time defined in the SLA. Thus, considering a variation in the number of clients, a developer could benchmark his applications to check if non-functional requirements of scalability, for instance, are met. Furthermore, the addition of machines was made manually, while our adopted definition of elasticity considers an autonomic management. [7] also presents the *scaleup* metric. This is not an elastic metric, since does not consider the variation of resources while running a workload. According to this metric, a system has good scaleup properties if the performance remains constant as the number of servers, amount of data and throughput scale proportionally.

3.1.1 Under-provisioning Penalty (*underprov*)

The first metric we propose is named *underprov*. The strategy for *underprov* is compliant to the concept of penalties for database services in the cloud. In this case, the resources allocated to the consumer are under-provisioned, i.e. insufficient to keep up the quality, generating a response time increase and consequently causing a penalty. This penalty is for violating the SLA. Penalties due to under-provisioning are directly related to bad elasticity, since, if the system had a good one, it would have identified the workload variation and would also have taken action to scale up to address the increase of demand. In addition, the time to add and remove machines is related with elasticity and this can be indirectly captured by measuring on how the SLA is satisfied. For instance, if a machine is quickly added when the system is overloaded, the system takes shorter time under heavy load and, consequently, the impacts over response times are reduced causing

less violated queries, i.e. more queries satisfy the SLA.

We define this metric in equation (3.1) as the average of the ratio execution time by expected time of those n queries whose response times are greater than the upper bound defined in the SLA and that are not discrepant values. We consider the fraction execution time by expected time in order to measure how far from the expected time, the execution time is. We argue that from an end-user point of view there are different perceptions of quality between violating the expected time by a very little amount of time and by a remarkable amout of time. In order to remove discrepant values, caused by timeout or instability peaks, for instance, we consider a percentile for under-provisioning. Values higher than the defined percentile are not considered by this metric calculation. By default, we consider 99^{th} percentile for *underprov* metric, to have a more precise metric.

Expected response time (*expected*) or SLA upper bound is defined by operation type in the SLA. This time represents the maximum time a query should take without disrespecting the SLA. The violated execution time ($violated_{et(i)}$) represents time spent by a query i that did not meet the SLA and that is lower than the defined *underprov* percentile.

$$underprov = \frac{\sum_{i=1}^{n} \frac{violated_{et(i)}}{expected}}{n} \tag{3.1}$$

The higher *underprov* is, the less elastic the database system is, because the system could not identify the need of adding more resources, could not quickly act to maintain acceptable response times and then more queries violated the SLA. $\frac{violated_{et(i)}}{expected}$ is always greater than 1, since it is applied only for violated queries, and measures the difference between defined and executed time.

In order to better understand on how to calculate *underprov*, consider the following hypothetical response times, in μs: [400; 150; 180; 250; 120; 300; 130]. For a matter of simplicity, consider also that discrepant values were already removed, making use of percentiles. If we assume $expected = 200\mu$s, the response times that violate the SLA, i.e. that are greater than the expected response time (SLA upper bound), are [400; 250; 300]. For these three violated queries, the values of fractions $\frac{violated_{et(i)}}{expected}$ are [2; 1,25; 1,5]. Finally, the *underprov* value is calculated as follows:

$$underprov = \frac{2 \ + \ 1,25 \ + \ 1,5}{3} = \mathbf{1,5833}$$

Apart from percentiles, we verified also other mathematical devices to identify discrepant values. One of them was interquartile analysis with outliers. Interquartile analysis identifies outliers in the data distribution by calculating the interquartile range (IQR) and defining upper and lower fences. Values are considered outliers if they are out of those fences. For instance, values are said to be extreme outliers if they are out of the range $[Q_1 - 3 * IQR, Q_3 + 3 * IQR]$, where Q_1 is the 1^{st} quartile, Q_3 is the 3^{rd} quartile and IQR is the difference $Q_3 - Q_1$ [43]. Even though interquartile analysis seems to be a good solution to identify divergent values and that avoids the definition of magic numbers, like 99^{th} percentile, this analysis may lead to unexpected behaviors. To illustrate these behaviors, in some experiments, interquartile analysis classified more than 12% of data as outliers, which is a remarkable amount of data and that could lead us to remove data that are not discrepant and that could represent a peak of clients in the experiments. In addition, if the upper fence value is lower than the SLA upper bound, all response times greater than the upper fence would be removed, our metric would result 0 (zero) and it would be given a false impression of perfect elasticity, although there were queries that may not have satisfied the SLA.

3.2 PROVIDER PERSPECTIVE

From a customer perspective, we just presented *underprov* metric. For a DBaaS provider, besides measuring that, it is also essential to evaluate how efficient the database system is to allocate only the minimum amount (or as closely as possible of the minimum amount) of resources to meet the SLA. Thus, from a provider perspective, our approach proposes *underprov* and a new metric named *overprov*, based on the charged level of resources. Finally, we combine *underprov* and *overprov* to get a single dimensionless value for elasticity of cloud database systems.

3.2.1 Over-provisioning Penalty (*overprov*)

When there is over-provisioning, the provider offers more resources than necessary to meet a demand. Thus, the provider is subject to an operating cost higher than the necessary to meet the SLA. In this situation, the database system has a number of resources that are running a given workload, but this amount may be higher than necessary. Unlike the penalty for under-provisioning, this metric does not make sense from a consumer perspective, since for a DBaaS consumer there is no problem to have more available resources if that does not imply in a cost increase.

overprov considers the execution time of queries performed when the database system is over-provisioned. We define this metric in equation (3.2) as the average of the ratio lower bound time by execution time for those m queries that are considered over-provisioned. In this work, a query is said to be over-provisioned if its response time is less than the lower bound and it is greater than such a percentile. We call this percentile of *overprov* percentile and the 1^{st} percentile is the default one. Unlike *underprov*, *overprov* moves the *expected* to the numerator since in an over-provisioning scenario the execution time is supposed to be lower than the expected one.

The query execution time ($query_{et}$) is the time spent by a query i that is considered in the *overprov* calculation.

$$overprov = \frac{\sum_{i=1}^{m} \frac{expected}{query_{et(i)}}}{m} \tag{3.2}$$

The higher *overprov* is, the less elastic the database system is, because the system kept more resources than necessary to satisfy SLA. $\frac{expected}{query_{et(i)}}$ is always greater than 1, since query execution times are always lower than expected time in an overprovisioning scenario.

As exemplified for *underprov* metric, to better understand *overprov* metric, consider the following hypothetical response times, in μs: [400; 150; 180; 250; 120; 300; 130]. For a matter of simplicity, consider also that discrepant values were already removed, making use of percentiles. If we assume $expected = 140\mu$s, the response times that violate the SLA, i.e. that are lower than the expected response time (SLA lower bound), are [120; 130]. For these two violated queries, the values of fractions $\frac{expected}{query_{et(i)}}$ are [1,1667; 1,0769]. Finally, the *overprov* value is calculated as follows:

$$overprov = \frac{1,1667 \ + \ 1,0769}{2} = \mathbf{1,1218}$$

Figure 3.1 illustrates the five possible ranges in which queries can be placed depending on their response times. Acceptance range is the interval in which queries are included neither on *underprov* nor on *overprov*. In the example presented for *underprov* and *overprov* metrics, the values 150 and 180 are within the acceptable range, because they are between SLA upper and lower bounds. Underprovisioning range contains queries whose response times are lower than the defined percentile (by default, 99^{th}) and greater than the SLA upper bound for response times, defined on the SLA for such an operation type. Overprovisioning range contains queries whose response times are lower than the SLA lower bound and that are greater than the *overprov* percentile (1^{st}, by default). Queries that are placed in percentile areas are discarded both from underprovisioning and overprovisioning metrics.

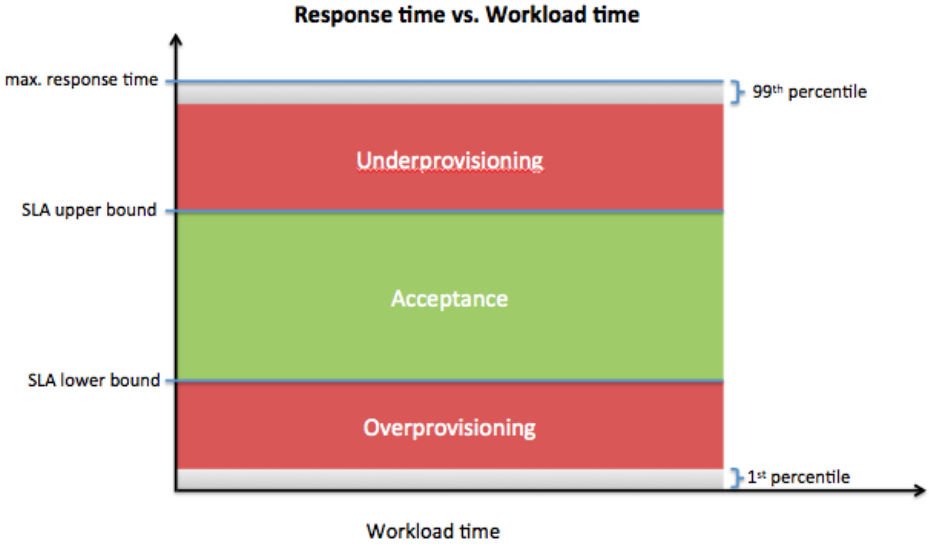

Figure 3.1: Time ranges where each response time can be placed within

3.2.2 Elasticity for Database System ($elasticity_{db}$)

The provider is impacted both on under- and over-provisioning scenario. Thus, from provider perspective, elasticity can be given by a value somehow composed of *underprov*

and *overprov* values. We claim that different weights should be applied to *underprov* and *overprov* when combined in a single metric and a higher weight should be set to *underprov* when compared to *overprov* weight for one reason. The penalty due to over-provisioning affects only one of the parties, i.e. the provider. Penalties due to under-provisioning imply in a cost for the provider, who will have to pay to or to offset the consumer. Besides, for DBaaS consumers, the quality of service for their clients will be compromised when the service response takes longer than expected.

Lets assume for a moment that a is the *underprov* weight and b is the respective *overprov* one. In our context, it is not necessary to define exactly which values each weight assume. We need to know only how greater than b the a value is, i.e. what is the value of a/b. Thus, for a matter of simplicity, we can set *overprov* weight to 1 and rename the fraction a/b to x.

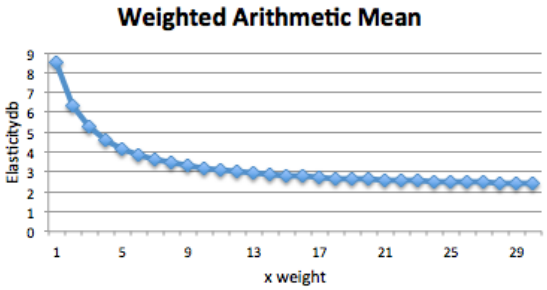

Figure 3.2: Metric values when adopted weighted arithmetic mean

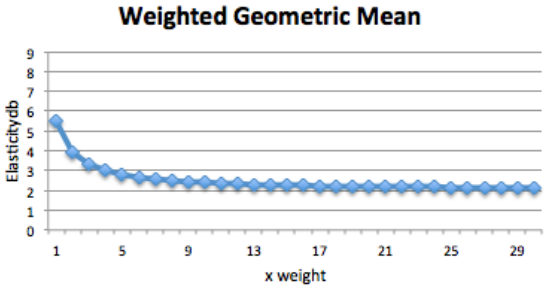

Figure 3.3: Metric values when adopted weighted geometric mean

In order to provide a dimensionless metric, named $elasticity_{db}$, that combines *underprov* and *overprov* metrics and that takes into account different weights for these metrics, we analyzed three weight functions: weighted arithmetic, geometric and harmonic means. To illustrate the fashion of each function, we fixed *underprov* to 2, *overprov* to 15 and varied x from 1 to 30. Figure 3.2, Figure 3.3 and Figure 3.4 plot charts of

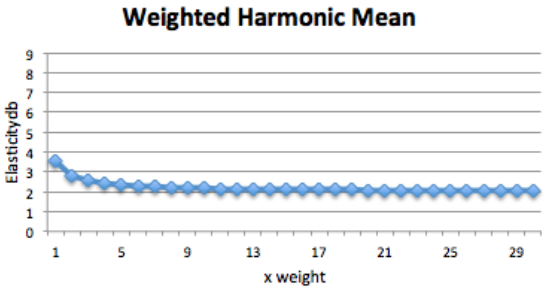

Figure 3.4: Metric values when adopted weighted harmonic mean

weighted arithmetic mean, weighted geometric mean and weighted harmonic mean, respectively. For the three kinds of weighted mean, the greater x is, the closer to *underprov* the $elasticity_{db}$ value is. We can notice that $elasticity_{db}$ values in geometric and harmonic charts vary between a short range, when compared to arithmetic chart. Thus, for a matter of simplicity on the calculation and to have a wider range of values to differentiate better elasticity of cloud database systems, we define $elasticity_{db}$ as a weighted arithmetic mean of *underprov* and *overprov*, as shown on formula 3.3. Numerical domain of x is $[1, \infty)$.

$$elasticity_{db} = \frac{x * underprov + overprov}{x + 1} \tag{3.3}$$

For instance, x weight could be defined based on costs. In this case, x could be defined by the cost of paying for underprovisioning penalties and *overprov* weight (set to 1) by the cost that could be saved if some resources had been released when environment was overprovisioned. Lower values of $elasticity_{db}$ indicate more elastic cloud database systems, since they make better use of resources while meeting the SLA. Our metric meets tests of reasonableness, such as (i) elasticity is non-negative and (ii) elasticity captures both over- and under-provisioning.

By defining *underprov*, *overprov* and $elasticity_{db}$ metrics, we have just proposed a model that can represent elasticity of cloud database systems. Therefore, we can now quantify elasticity of such systems. After fulfilling the requirement of defining a model for elasticity, we can now propose a benchmark tool to properly measure this characteristic of cloud database systems.

CHAPTER 4

BENCHXTEND

We propose a benchmark tool called BenchXtend [6] which extends YCSB. YCSB was chosen among other possible benchmark tools mainly because it already has connectors for many NoSQL database systems as well as a JDBC driver can be user for relational systems, it is open source and it is designed in such a way that allows to extend it. This tool provides a way to change the number of clients while running a workload, as well as to calculate the metrics proposed in this work. Varying the load of a system is essential to properly evaluate its elasticity by stressing the system in such a way it can react to maitain quality of service. The load variation for database systems can be performed basically in two ways: (i) keeping the number of clients but changing the throughput by client, or (ii) changing the number of clients but keeping the throughput by client. YCSB does not implement any variation on the load, that means the expected throughtput by client and number of clients remain the same throughout the experiment. BenchXtend implements the change on number of clients both increasing and decreasing during workload execution, since this approach illustrates a more realistic scenario where users access and leave applications all the time.

Figure 4.1 presents the architecture of our solution. The *Client Manager* component controls the variation of clients. *Metrics* component is responsible for calculating *underprov*, *overprov* and $elasticity_{db}$ metrics after a workload execution. These metrics are computed by operation type and according to some parameters passed as input. After executing all queries and calculating elasticity metrics, *Exporter* component outputs the query response times and metrics values.

The number of clients is changed automatically according to the timeline file. In our context, a timeline is simply a list of pairs $< time,\ number\ of\ clients >$ explicitly defined that describes the expected number of clients in such a moment. Depending on how timeline is defined, there can be a considerable gap between two timeline entries. Thus, we interpolate the number of clients between two timeline entries by implementing two functions: Linear and Poisson. Linear function calculates intermediate values for each

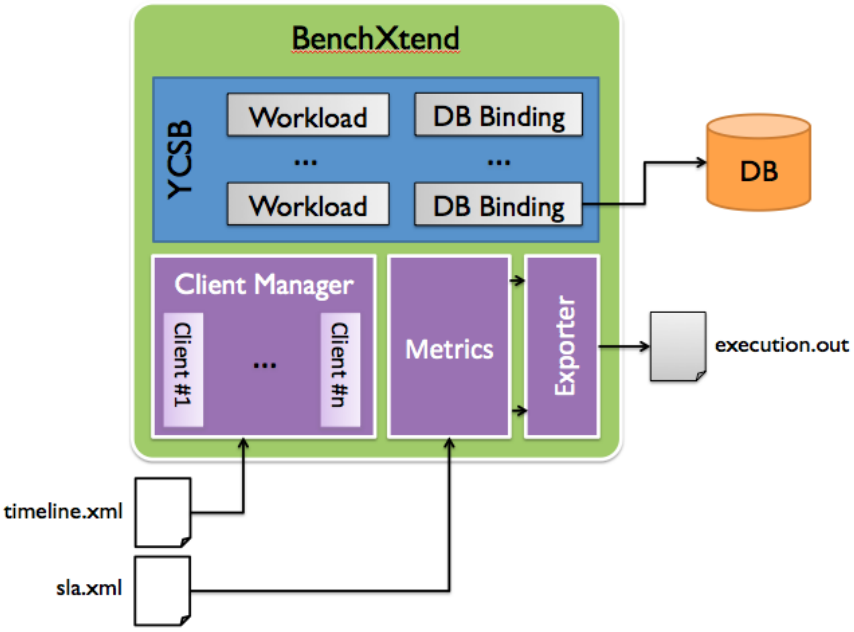

Figure 4.1: BenchXtend architecture

second, as its name suggests, in a linear fashion from the initial to the subsequent entry. When selected Poisson function, the lambda (λ) parameter, that is equals to the variance of Poisson distribution, adopted is always the second value. For instance, if interpolating from 5 to 11, λ chosen is 11. Even though we provide Linear and Poisson implementations, the architecture is designed to allow a user to implement his own distribution. Figure 4.2 shows the original timeline and figures 4.3 and 4.4 present examples of Linear and Poisson interpolations for the original timeline, respectively. In our experiments, we prefered to use linear interpolation to avoid abrupt increase or decrease on the number of clients.

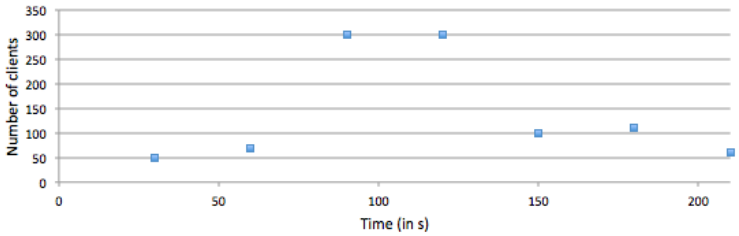

Figure 4.2: Timeline without any interpolation

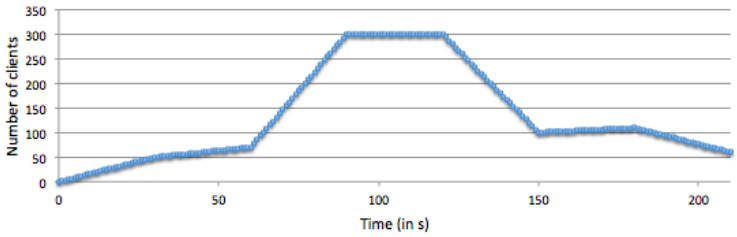

Figure 4.3: Timeline with Linear interpolation

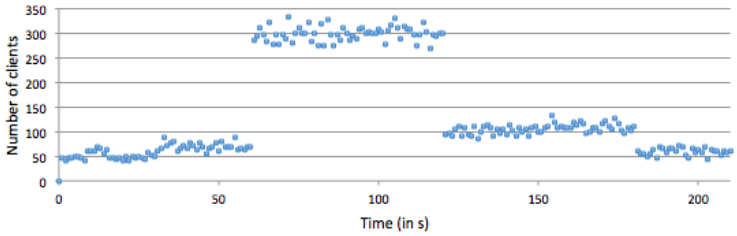

Figure 4.4: Timeline with Poisson interpolation

4.1 MONITORING AND SCALING IN AND OUT

Our BenchXtend tool sends queries to a cloud database system (see Figure 2.1). Since our focus is not on how well or badly designed the cloud database system itself is, but on the benchmark tool, for a matter of simplicity, we implemented our own Instance Manager [44] in Ruby making use of Amazon EC2 API. Other authors [34] [35] propose some solutions to manage instances. [34] presents a systems that not only adds and removes nodes, but also reconfigures them in a heterogeneous manner according to the workload's access patterns. [35] provides an implementation of a decision-making module as a Markov Decision Process, enabling optimal decision-making relative both to the desired policies as well as to changes in the environment the cluster operates under. Even though [34] and [35] seem to be good solutions for this purpose, they are not available to be deployed in Amazon EC2 environment and they would require additional research that is out of scope for this work. Figure 4.5 illustrates how BenchXtend and our Instance Manager act and how they interact with the cloud platform. Figure 4.5 represents the environment instantiated to evaluate our tool and our metrics. If we compare Figure 2.1 and Figure 4.5 we can notice there is no Database Manager in the later figure. This is due to the decentralized characteristic of the adopted database system, Cassandra. Instead of dedicating a node to a Database Manager, we use that node as a regular data node.

Our Instance Manager is composed of a Monitor and a Decision Taker. Monitor

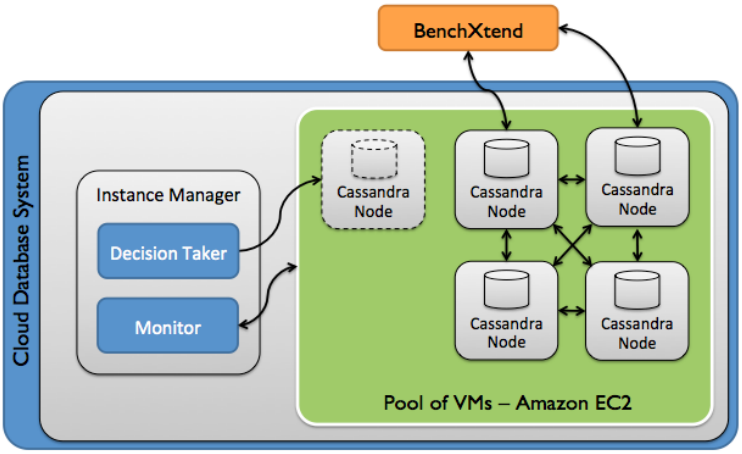

Figure 4.5: Cloud database system instantiated for our experiments and BenchXtend

collects CPU usage from each running instance, via a SSH command, on every 5 seconds and saves it into a file. Decision Taker executes on every 60 seconds, that is time enough to have at least 10 new entries generated by the monitor, and then reads the last 10 entries of all files (one for each running machine). For each file, if 7 out of 10 entries exceed the maximum CPU usage threshold, that was set to 60%, we increment an *add_machine* counter. After analyzing all files, if $add\_machine \geq \frac{machines\_running}{2}$, we add a machine. Similarly, if 7 out of 10 entries are lower than the minimum CPU usage threshold, that was set to 20%, we increment the *remove_machine* counter. If $remove\_machine \geq \frac{machines\_running}{2}$, we remove a machine. In both cases, *machines_running* is the number of data instances running at that moment. The fraction $\frac{machines\_running}{2}$ is used to define that only if at least half of the running machines are overloaded (or underloaded) the Instance Manager acts to adapt the nodes. This avoids that a sudden variations on only one or few (less than half) machines fire the action to add or remove an instance. For the number of running machines, we have lower (2 machines) and upper bounds (4 machines). If the Instance Manager decides to remove a machine and there are only 2 machines running, the removal process is then canceled and running machines remain the same. Similarly, if it is decided to add a new node but there are 4 machines running, the addition process is canceled. Upper and lower bounds of machines are necessary to deal with constraints of the experimental environment and they can be configured before starting the Instance Manager.

The values for CPU thresholds were defined based on the values adopted by [12]. [12] adopts in one of its experiments 70% of average CPU usage as the limit to add

a machine and 20% to remove one. Initially, we set our threshold to 70%. However, we realized that to reach this CPU usage we needed to increase the number of clients and Cassandra started throwing more exceptions and refuse more connections. Thus, we relaxed a little bit our threshold from 70% to 60%. [12] monitors CPU with one minute of interval. We reduced this time to 5 seconds in order to collects more usage entries and to have a quicker response for demand fluctuation. Even though have adopted these values for thresholds and intervals, they can be easily configured before starting the Instance Manager. [39] proposes a Transaction/Workload Monitor that altogether with an Action Manager and an SLA checker act conceptually as our Instance Manager. [12] also proposes a component, called autoscaling engine, that monitors and takes decision on adding or remove machines.

Apart from our Instance Manager, on [44] we provide a set of Ruby and Shell auxiliary scripts that allow to:

- calculate elasticity metrics using the output of a BenchXtend execution
- calculate histogram of response times based on BenchXtend output
- treat BenchXtend output to filter response times by operation type

Firstly, with these scripts it is possible to recalculate elasticity metrics without requiring to re-execute a workload. This is very useful and save lots of time to figure out suitable values for parameters like SLA upper bounds and *underprov* weight (x). Secondly, given the output of a workload generated by BenchXtend, we created a script to generate the histogram of response times to better analyze the data distribution shape. Properly understanding the data distribution shape can help providers on analyzing the workload results. Finally, we provided a script to filter response times making easier to retrieve desired data from a file that have more than 1GB and more than 30 million lines, depending on the workload duration. All these scripts aim to minimize the effort of evaluating workload results and dealing with a large amount of output data.

4.2 YCSB EXTENSIONS

Some changes were implemented in YCSB code to implement the features of BenchXtend. These extensions were made in such a way to maintain the core of YCSB avoiding as

much as possible to include overhead processes in BenchXtend. The following changes were made:

- Developed the reading of an XML input file with SLA rules
- Developed the reading of an XML input file with the timeline entries and a set of parameters to calculate elasticity metrics
- Extended a class to perform measurements during workload and export the gathered results
- Implemented a manager to control the addition and deletion of clients that send queries to databases
- Adapted Cassandra connector
- Adapted the moment when workload configuration files are read

Firstly, two new files are expected to be provided when executing a BenchXtend workload. The first file (sla.xml) defines the SLA with the expected response times for each operation type. Upper and lower bounds are defined in microseconds (μs). The upper bound defines the maximum time a query should take to not to be elegible to be an under-provisioned query. Similarly, the lower bound defines the threshold whose queries with lower response times could be considered over-provisioned. There are no hard and fast rules to specify expected response times for SLA, since they tightly depend on the environment where the workload is executed. The second file (timeline.xml) defines the timeline, that describes the variation of clients by time. In addition, this file defines the interpolation function to be used among timeline entries and the values of the parameters explained on Table 4.1. These parameters are used when calculating elasticity metrics.

We also implemented a new class called *IndividualMeasurement* that extends the *Measurement* base class. This new class is responsible for measuring query response times as well as for calculating elasticity metrics and maximum, mininum and average response times. Besides, this class prints response times of each executed query. Since the amount of queries can be huge (about hundreds of thousands), in order to provide a better way to plot the results, we also calculate average response time by second, i.e. sum up all queries **started** in such a second and divide by the total number of queries summed up. In addition, this class also calculates some statistics metrics to characterize

Parameter	Value	Description
underprov weight (x)	float, $x \geq 1$	defines a weight for *underprov* metric to calculate $elasticity_{db}$ metric
underprov percentile	float, $0 < percentile < 1$	defines a percentile value used to trim the highest and discrepant response times
overprov Percentile	float, $0 < percentile < 1$	defines a percentile value used to trim the lowest and discrepant response times

Table 4.1: Parameters included in the timeline.xml file to be used for elasticity metrics

the data distribution of response times. Mean, median, quartiles, interquartile range, minimum, maximum, 1^{st} percentile and 99^{th} percentiles are calculated and printed when finished the workload execution. With these measures, one could create a box plot to have a graphical representation of the distribution.

The most remarkable change was related to how to manage client threads to execute operations. On the original YCSB, the number of operations is defined by the *operationcount* property and the number of threads by the *threadcount* one. Basically, operations are evenly distributed among all threads, i.e. all threads are started in the beginning of the workload and each thread executes (*operationcount* / *threadcount*) operations. Thus, there is no variation on the number of clients (threads) sending queries to the database system. Each thread holds a database connection with a database node randomly chosen from the available ones in the cluster. Unlike regular YCSB, in our approach the number of threads varies according to the input timeline. We developed a Client Manager that is responsible for managing clients, by creating or stopping them. Since our approach is based on a time series instead of a number of operations to be performed, on BenchXtend, threads keep executing queries consecutively while a stop requests are not sent to them by the Client Manager. When the Client Manager sends a stop request a to a thread, if the client is executing a query in that moment, the tool waits for that query to return a result and then stops the client not to send queries anymore. As a throttling approach, we set a sleep time of 500ms between the execution of two operations of the same thread. As commented in YCSB code, this is more accurate than other throttling approaches YCSB developers have tried because it smooths timing inaccuracies over many operations. If changing this sleep time, the number of operations performed by BenchXtend is inversely affected, i.e. decreasing this time implies in more

queries sent to the database system.

Cassandra does not have a central node, or a set of nodes, responsible for making the interface with client applications and for redirecting operations to the appropriate data nodes. This central node is usually responsible for balancing the load and for managing nodes that join or leave the cluster. MongoDB, for instance, has Query Routers nodes that play this role, sending operations to shards (data nodes, considering the cluster is sharded). Since Cassandra does not have this central node, Cassandra DB Binding should act as load balancer and manage when new nodes join or leave the the cluster. However, it does not act like this because it was not originally designed to deal with a variation of resources (VMs) of a cloud database system during workload execution. Therefore, we had to adapt the original Cassandra DB Binding to balance the load.

In YCSB, the hosts are fixed and defined before starting the workload. However, for Cassandra, it is crucial to let the benchmark tool know when any machine is added or removed. Thus, BenchXtend addresses the problem of sending or avoid sending queries to machines just added or just removed by reimplementing Cassandra DB Binding of YCSB. Our strategy is to make Instance Manager updates the *hosts* property of then workload input file when the cluster changes. Periodically, our DB Binding reloads the list of hosts to see if new hosts were added or if existing ones were removed. On every ten queries sent by a client, Cassandra DB binding establishes a new database connection with other randomly-chosen host from the cluster. Thus, when a new node joins the cluster in few seconds (about 15 seconds, depending on how long it takes to execute up to ten queries) the benchmark tool starts trying to connect to it. When a node leaves the cluster, queries that are being performed on the leaving node may fail or time out. In this case and in any other case when an exception is thrown, a new connection is established with other available host.

Ideally, it would be better to have a separate component of the benchmark tool to interact with Instance Manager to remove an instance only when there is no query being performed. In this way, Instance Manager would warn this component that a node is about to be removed. This component would work to avoid sending queries to the machine that is going to be removed, moving the queries to other machines. When there were no thread sending queries to that node, this component could response to the Instance Manager saying that the node could be safely removed. Thus, the number of failed or time out queries due to node removal would be zero or almost zero. Although it presents remarkable advantages, it would require architectural changes that could impact

on all DB Bindings, not only on the Cassandra one. Due to a time restriction, this feature is presented as a suggestion for future work.

BenchXtend relies on YCSB to define on how queries are built and selected. YCSB must make many random choices when generating a load, like which operation (read, insert, update, delete or scan) to perform , which record to read or write, how many records to scan, and so on. These decisions are governed by random distributions. YCSB has several built-in distributions like uniform, zipfian, latest and multinomial. BenchXtend does not extend any of those distributions. Distributions can be configured via workload configuration file by changing the *requestdistribution* property.

Even though we propose some changes to calculate elasticity metrics, one can take advantage only of our feature to vary of number of clients and keep gathering regular results of YCSB. YCSB allows to output the workload results as a time series or a histogram. Therefore, BenchXtend can also be used to perform workloads and gather common performance metrics, while the number clients varies during workload execution.

4.3 COMPARING BENCHXTEND AND RELATED WORK

In order to compare our tool with other related tools and summarize their differences, we propose the Table 4.2. Trying to make a fair comparison, we also added in the comparison features that are present in some related tools and that BenchXtend does not have. We have chosen for this comparison TPC-C [45], from TPC family, OLTP-Bench and YCSB. TPC-C was chosen because it is a well-known and very used benchmark for OLTP database systems. Cells filled with an 'x' letter mean the tool of the corresponding column has that feature of the row. Since OLTP-Bench embeds YCSB as one of its benchmarks, all features present on YCSB are also present in OLTP-Bench. Similarly, since BenchXtend is an extension of YCSB and we did not remove any feature of this later, all features of YCSB are also present in BenchXtend. BenchXtend presents the capacity of calculating elasticity metrics as a differential when compared to all other listed tools. Since BenchXtend is an extension of YCSB and relies on the latter to define which operations are selected to be performed, BenchXtend does not support the execution of complex queries or transactions with multiple queries, but this is an explicit design choice of YCSB. YCSB does not attempt to exactly model a particular application or set of applications, as is done in benchmarks like TPC-C. Such benchmarks give realistic performance results for a narrow set of use cases. In contrast, YCSB goal is to examine

a wide range of workload characteristics, in order to understand in which portions of the space of workloads systems performed well or poorly [7]. Benchmark tools can also be classified in synthetic or empirical, according to how the used data is generated. Synthetic benchmarks emulate typical applications in a pre-determined problem domain and create a corresponding synthetic workload. Empirical benchmarks utilize real data and tests and re-invent the actual database applications [46]. None of the analyzed tools use real data. OLTP-Bench makes use of real schemas, but the data is synthetic.

Even though OLTP-Bench presents more features than YCSB, we decided not to extend OLTP-Bench because its architecture was designed to be extensible by adding new benchmarks, but changing the way clients are managed would require an effort that would turn unfeasible the planned extension. In addition, the YCSB source code is much better documented and comprehensible than OLTP-Bench code.

Feature	TPC-C	OLTP-Bench	YCSB	BenchXtend
Benchmark relational systems	x	x	x	x
Benchmark NoSQL systems		x	x	x
Variation of load during execution		x		x
Calculation of elasticity metrics				x
Execution of complex queries	x	x		
Synthetic workloads	x	x	x	x
Empirical workloads				

Table 4.2: Comparison of some benchmark tools for database systems

Our BenchXtend tool presents useful extensions that can improve the way cloud database systems are evaluated. Altogether with the elasticity model discussed on last chapter, the load variation provided by BenchXtend allow us to measure elasticity by performing some experiments and analyze their results.

CHAPTER 5

EVALUATION

The main goals of the experiments of this dissertation are (i) to validate our BenchXtend tool showing that it works as explained and (ii) to validate our elasticity model of cloud data systems. Thus, at least is this work, we do not aim to compare elasticity among database systems to stating which system is more elastic.

A differential of our work is to consider a dynamic pool of resources, where machines are added or removed during workload execution. Thus, in our experiments we compare results in two scenarios: (i) with elasticity, where the Decision Taker scales in and out and (ii) without elasticity, where no machine is added or removed during execution.

5.1 ENVIRONMENTS

In order to achieve the goals of our experiments, we consider that BenchXtend tool sends queries to a cloud database system with our own Instance Manager, one NoSQL database system and a pool of data nodes running on Amazon EC2 instances, as shown on Figure 4.5. Cassandra (version 1.1.12) was chosen as the database system, due its wide adoption both by academy and industry [7] [47] [48] .

The following machines were used in our experiments: 1 EC2 instance (m1.medium - 1 vCPU, 2 ECU[1], 3.75 GiB[2] of RAM) to run BenchXtend tool; 1 EC2 instance (m1.small - 1 vCPU, 1 ECU[1], 1.7 GiB[2] of RAM) to run Instance Manager that gathers monitoring data from other nodes and takes decisions on adding or removing nodes. Moreover, we set up 4 instances (m1.large - 2 vCPU, 4 ECU[1], 7.5 GiB[2] of RAM) for Cassandra. Two Cassandra instances were set as seeds and the other two as regular data nodes. Seeds are

[1]One EC2 Compute Unit (ECU) provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor

[2]The gibibyte (GiB) is a unit of digital information storage. It is a binary multiple of the byte obtained using the prefix gibi (Gi). 1 gibibyte = 2^{30} bytes = 1073741824bytes = 1024 mebibytes.

started before executing the workload while regular data nodes keep turned off until the Decision Taker starts one of them. All machines were placed in the same Amazon EC2 zone (us-east-1c) and none of them were EBS-optimized.

Some Cassandra properties were changed on cassandra.yml configuration file to perform our experiments. *thrift_framed_transport_size_in_mb* was changed from 15 to 240 and *thrift_max_message_length_in_mb* from 16 to 256, to avoid problems when perfoming batch inserts on load phase and large scan queries. *rpc_timeout_in_ms* was increased from 10000 to 30000 in order to avoid or minimize inter-node communication timeouts when the system is under heavy load. Apart from Cassandra configurations, we had also to increase ulimit[3] on Linux, since the number of threads (one by client) created during workload goes beyond the Ubuntu default value (1024) for open files descriptors. We followed DataStax Troubleshooting Guide [49] to properly set ulimit[3] values.

5.2 EXPERIMENTS

YCSB provides by default 5 Core Workloads described on Table 2.2. In our experiments, we performed Workload A (Update heavy - 50% read, 50% update) and Workload E (Short ranges - 95% scan, 5% insert). Thus, with these two workloads, we cover 4 out of 5 available operation types. Database was populated with 4 millions 1-KB 10-field records, generating a database of about 4GB of size. Workload was executed for 4 hours, the number of clients was changed according to a timeline and a Linear function was chosen to interpolate timeline entries. Iniatially, each workload was executed during 1 hour. However, due to the long time to add Cassandra nodes, we increased the duration to 4 hours. Taking more than 4 hours by workload was unfeasible due to budget constraints, since the experiments were performed on Amazon EC2 environment. For workload A we use the timeline illustrated on Figure 5.1. For workload E we use a different timeline shown on Figure 5.13, because the number of clients of workload-A timeline was too high to perform scan queries of workload E.

Regardless of the workload, initially, we start Cassandra seeds and then perform the BenchXtend load phase to populate the database. After that, we restart seeds machines and then start Monitor and Decision Taker. Before starting the run phase of the benchmark, we check if no compaction of Cassandra SSTables is being performed. During

[3]ulimit is a Linux command that limits the use of system-wide resources, like the maximum number of open file descriptors.

Operation	Upper bound (in μs)	Lower bound (in μs)
Read	200000	50000
Update	100000	30000
Insert	100000	30000
Scan	700000	200000

Table 5.1: Expected response times for each operation defined in the SLA

compaction [50] there is a temporary spike in disk space usage and disk I/O that may affect our results. Instances are added by the Instance Manager during workload, based on the variation of the monitored CPU usage, as described in the Chapter 4.

Each added instance has Cassandra already configured, but with an empty database. Our experiments relied on Cassandra to manage consistency in each replica, as well as to stream data when nodes are added or removed. Replication factor was kept the default, i.e. 1, that means there is only one copy of each row on the cluster. Consistency level considered was also the default, i.e. ONE, that means an operation must succeed in at least one replica before returning an acknowledgement to the client application. As soon as a node is added or removed, Instance Manager updates, in the VM where BenchXtend is running, the *hosts* property of the BenchXtend workload file to let it know about the cluster change. Updating this property is mandatory for Cassandra because it does not have a central node to where all workload queries are sent. Consequently, the benchmark tool has to know when the cluster has changed in order to deal with balancing the queries among the available hosts.

In the SLA file, we set expected response times by operation type: read, update, insert and scan. Table 5.1 shows the values for each operation. x weight was set to 5, but it can be configured before running a workload. These values were defined after some experiments in our environment and, so far, there is no rule of thumb to define them, althought it is reasonable to have x few times greater than 1.

5.3 RESULTS

Firstly, we present the results for each workload. After that, we select one workload and show what are the impacts on elasticity metrics when we vary some parameters. Finally, we present an overall analysis about all results, problems faced during experiments and define what conditions must be satisfied when comparing elasticity of two different

database systems.

5.3.1 Workload A

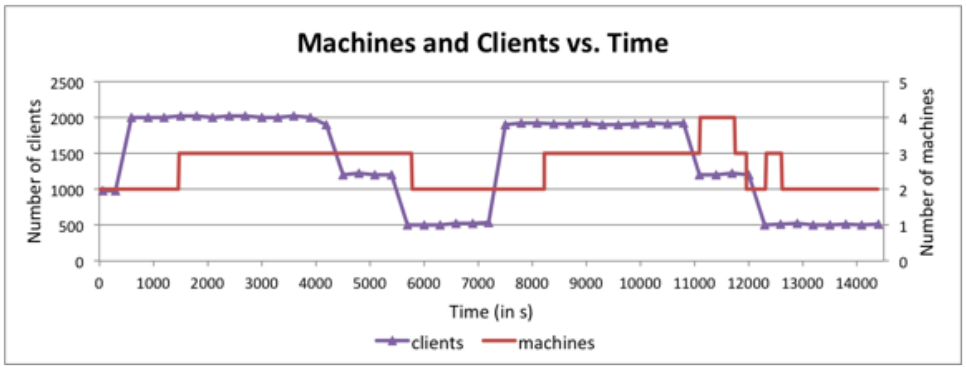

Figure 5.1: Scenario with elasticity

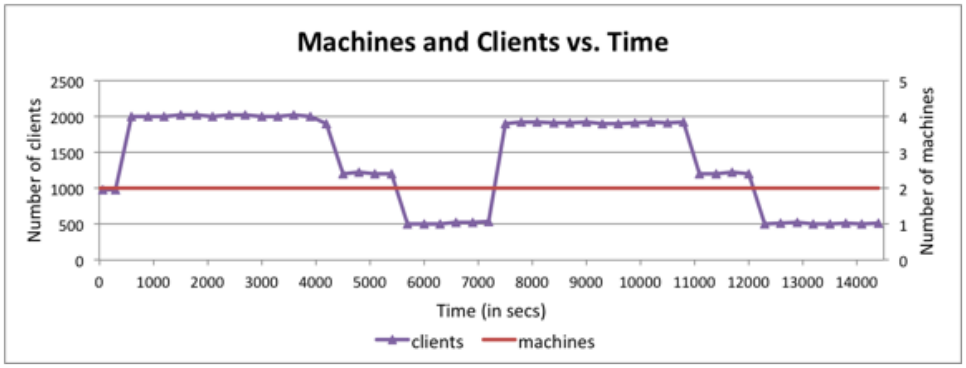

Figure 5.2: Scenario without elasticity

The chart of Figure 5.1 plots when machines are actually added or removed and compare them with fluctuation of number of clients defined in the timeline file that is read by BenchXtend before actually sending queries to the database system. The time to add (bootstrap) a Cassandra node may be considerably high and may vary a lot. To add the 1^{st} machine 3min were taken, while 12min to the 2^{nd} one, 4min to the 3^{rd} one and 4min to the last one. The average time to add a machine in this experiment was about 5-6min. In other experiments we could face more than 30 minutes to add a machine. A reason for that is the cost of data streaming to move data among nodes. When we add a new Cassandra node, this node assumes a token and consequently it owns a data range that was responsibility of one or two existing nodes. Thus, the nodes that "lose" part of their range stream data to the new node. This task may take several minutes, depending

on the amount of data to be transfered, on the number of SSTables and on the resources usage, like CPU and memory, in that moment.

Cassandra versions prior to 1.2.x require to evenly set *initial_token* for all nodes in the ring to have a perfect data distribution. It is also possible to let Cassandra decide how to split the data range, setting *auto_bootstrap* to *true*. The last option may incur in the risk of having hot spots, i.e. nodes receiving more queries than other ones, since there is no guarantee that the range will be evenly partitioned. In our experiments, we adopted the first option because it evenly divides the data range. *initial_token* was set for each node. For new nodes, the token is set to be in the middle of two tokens to avoid *move* operations that are known to be resource expensive. We do not execute *cleanup* operations during workload to remove keys no longer belonging to nodes that streamed data to the new one. Cleanup may be safely postponed for low-usage hours [51]. From versions 1.2.x, Cassandra provides vnodes feature that does not required to set *initial_token* anymore and that may reduce the recover and bootstrap time. After preliminary tests with version 1.2.6, we faced recurrent hang problems that led us to keep using version 1.1.12 in our experiments.

Another reason for the variation of bootstrap time is that Amazon EC2 presents performance issues depending on the moment the experiments are executed and on the CPU model of the physical machine where the VM is placed. Amazon EC2 instances may present noisy neighbor problem. In a noisy neighbor scenario one or more virtual machines on the physical host are consuming very large amounts of disk I/O resulting in very poor performance for the remaining virtual machines. EBS-optimized instances may minimize this problem by guaranteeing a higher number of IOPS. Other authors [52] [53] present further analysis about performance variations on Amazon EC2 instances. During our experiments we could identify a wide range of CPU models, like Intel Xeon E5-2650 and Intel Xeon E5507. If we compare these two CPU models [54], for instance, we can see that E5-2650 presents twice the number of cores and has better performance by core, what may reduce the noisy neighbor problem when compared to E5507, under the same load.

To remove (decommission) Cassandra nodes from the cluster, the operation time variation was lower, varying from 2min30s to 3min30s. We can notice a remarkable difference between the bootstrap time and the decommission time in our experiments. According to our analysis, when a node is bootstraped, the system is under a heavy load, since our Decision Taker adds a machine only when CPU usage is high. Under heavy

load, data streaming process takes longer since this is an I/O-intensive operation and the number of queries being sent at that moment is high due to the high number of clients. When a node is being decommissioned, the situation is opposite. The system is not under heavy load and, consequently, the impact on I/O requests due to client queries is low. To validate our analysis, we manually added and removed nodes, similarly to what Decision Taker did on Workload A as illustrated in Figure 5.1, but in a scenario when no query was being sent by BenchXtend. In such scenario, all bootstrap and decommission operations took from 2min to 3min30s. In conclusion, under heavy load, bootstrap takes longer and this fact should be considered when designing a strategy for Decision Taker, as well as when deciding if Cassandra is suitable for short-term variations.

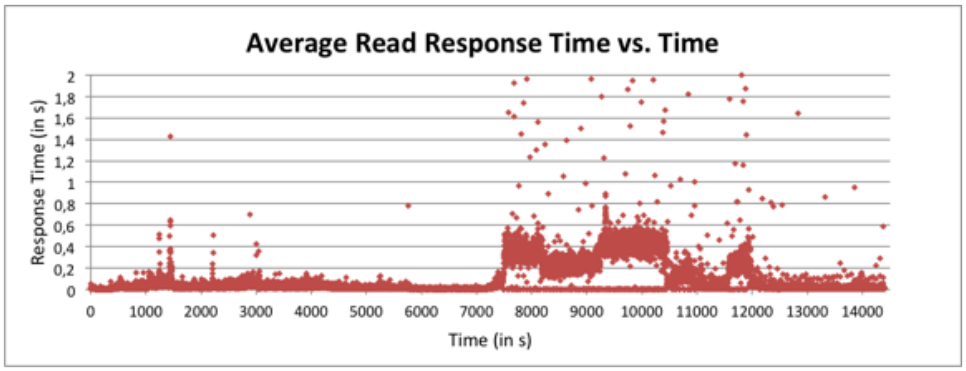

Figure 5.3: Scenario with elasticity

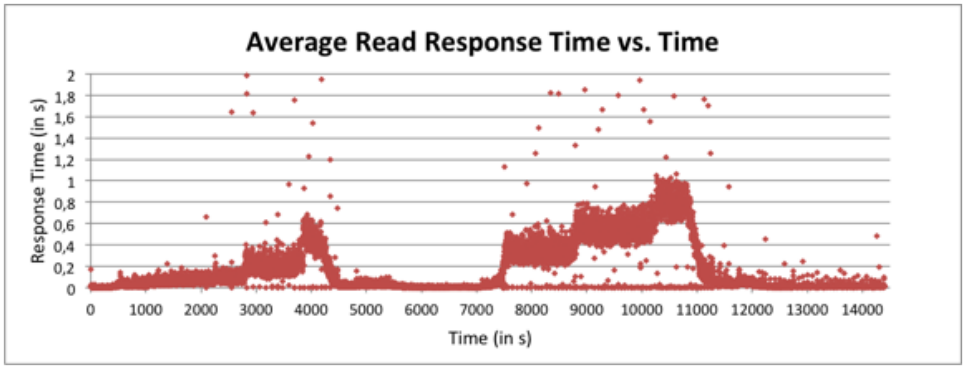

Figure 5.4: Scenario without elasticity

Since the number of queries executed by BenchXtend was very high (about 16 million queries by operation), we plotted the average response times instead of the individual response times. Figures 5.3 and 5.4 show the average response time by second, i.e. the average of response times for all read queries that started in a specific second, in the elastic and inelastic scenarios, respectively. Similarly, Figures 5.5 and 5.6 show average response times, but for update operations.

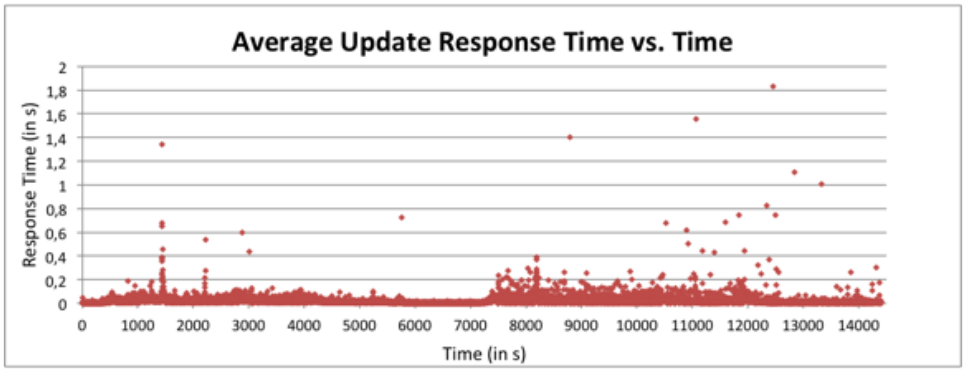

Figure 5.5: Scenario with elasticity

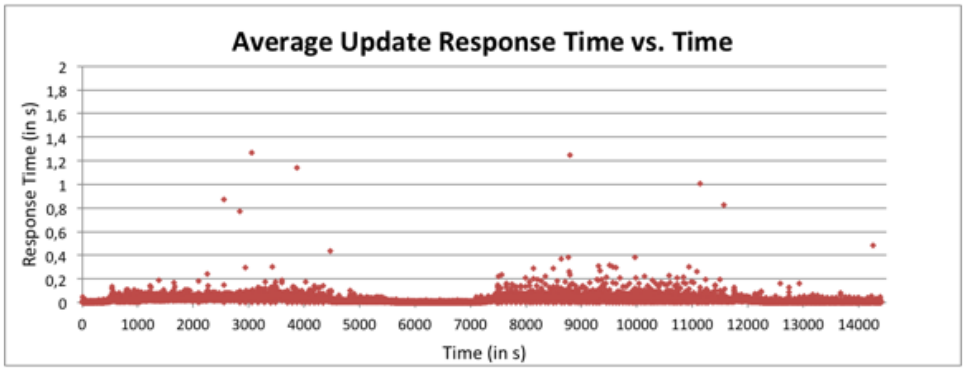

Figure 5.6: Scenario without elasticity

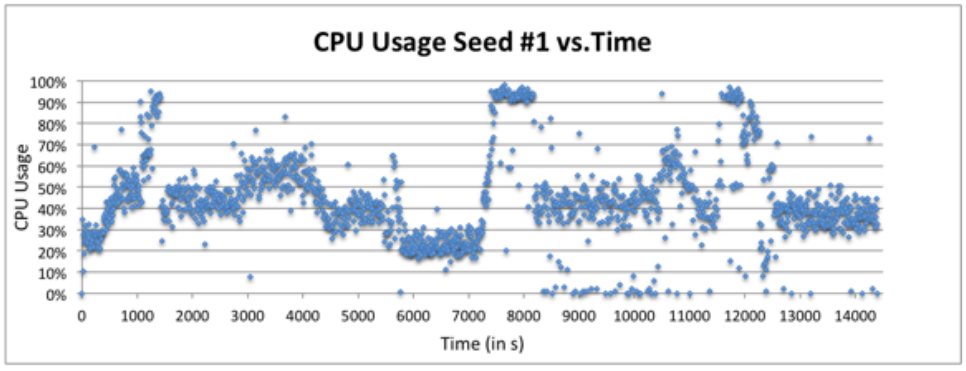

Figure 5.7: Scenario with elasticity

On workload A, as well as on E, the Instance Manager monitors CPU to take decisions on adding or removing machines. On Figures 5.7 and 5.8 we plotted CPU usage of Cassandra seeds #1 in the elastic and inelastic scenarios. Similarly, Figures 5.9 and 5.10 show CPU usage of seed #2. For both seeds, it is fairly reasonable to assert, based on the comparison with Figures 5.1 and 5.2, that CPU usage follows timeline trend with some variations when machines are added or removed, which is an expected behavior.

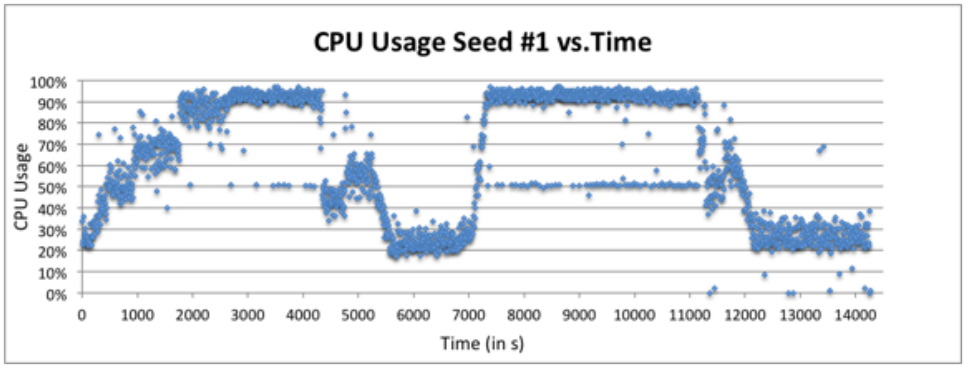

Figure 5.8: Scenario without elasticity

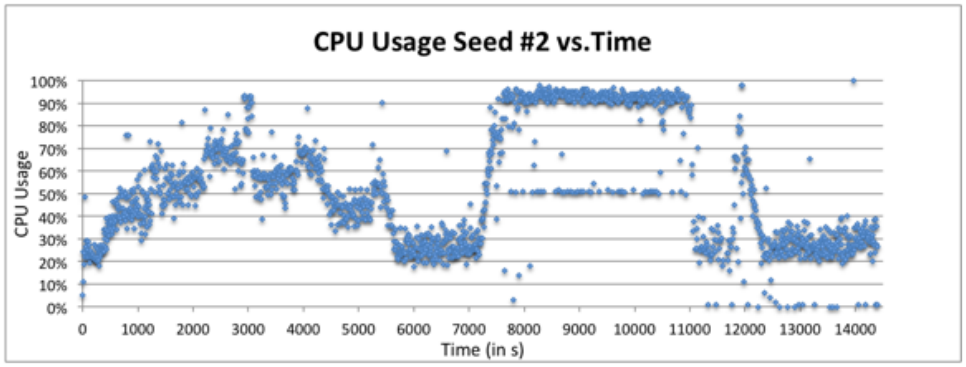

Figure 5.9: Scenario with elasticity

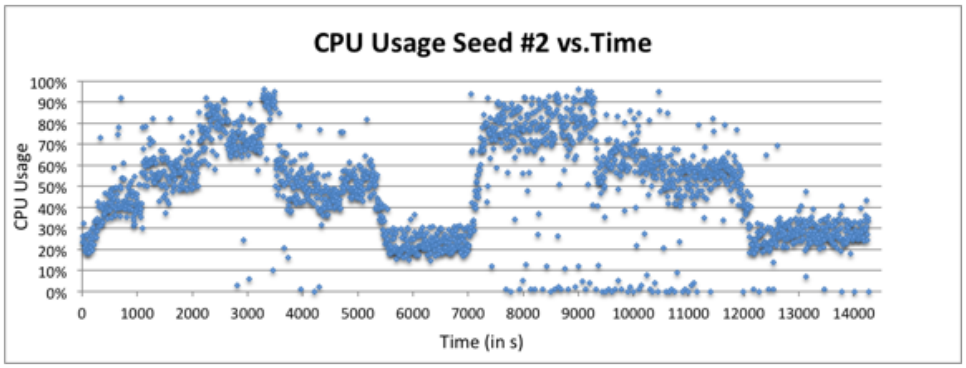

Figure 5.10: Scenario without elasticity

Thus, varying the number of clients during workload affects directly the indicator we are using to take decision on adding or removing instances.

Even though we can see that CPU usage follows the trend of variation on number of clients, it is more important to us to check if response times follow the same trend, because our elasticity model is based on how far from SLA response times the measured response times are. On Figures 5.3 and 5.4 we can see an increase on response times

	Read operation	
	with elasticity	without elasticity
Minimum (in μs)	362	362
Maximum (in μs)	31390923	63450637
Average (in μs)	133744.08	223995.20
Median (in μs)	9563	32235
1^{st} percentile (in μs)	856	881
99^{th} percentile (in μs)	1076529	1578348
1^{st} quartile - Q1 (in μs)	2943	4249
3^{rd} quartile - Q3 (in μs)	66421	189605
# of underprov queries	2165155	3637293
# of overprov queries	11851768	8470567
Total of queries	16839414	15625044
Underprov metric	2.621841	3.150996
Overprov metric	14.158379	13.677754
Elasticitydb metric	4.544598	4.905456

Table 5.2: Metrics calculated for Read operations

when the number of clients increases. On Figures 5.5 and 5.6 there is also an increase but it is not as noticeable as on the Read charts, because Cassandra is write-optimized and the amount of data into an update response is smaller than in a read one. Therefore, the response times for write operations are expected to be lower than the ones for read operations.

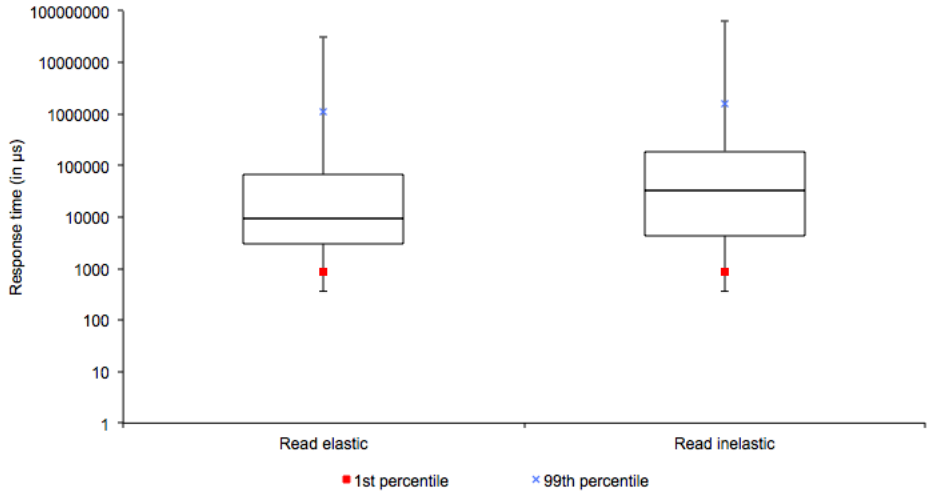

Figure 5.11: Box plot chart showing elastic and inelastic scenario of Read operation - Workload A

Metrics results calculated by BenchXtend for read operations of Workload A are shown on Table 5.2. In addition, Table 5.2 shows some values to characterize the data distribution, like minimum, maximum and quartiles. To have a graphical representation of the distribution, we provide also a box plot chart on Figure 5.11. The chart on Figure 5.11 uses logarithmic scale with base 10 for y axis to have a better view, since the maximum value is much higher than the 3^{rd} quartile. Whiskers values in the box plot are maximum and minimum response times. In addition, 1^{st} and 99^{th} percentiles are plotted. Analyzing both the table and the chart, we can see that the distribution is long-tail, since most data are closer to minimum than maximum value.

For *underprov* of read operations our model could capture an improvement when adding machines while system was overloaded. In this case, the number of violated queries on elastic scenario represents 59% of violated ones in the inelastic scenario. *underprov* had a lower value (2.621841) with elasticity, as expected. The lower this value is, the more elastic the system is. Unlike what was expected, *overprov* of read operations showed a higher value for the elastic experiment. This may suggest that the strategy adopted by Decision Taker to drop instances should be improved to release earlier machines when system is overprovisioned or to release quicker. It may also suggest that adding a machine may be more than enough to supply the demand. Although the *overprov* is higher in the elastic scenario, since we have different weights for each metrics, $elasticity_{db}$ for elastic scenario was lower, as expected. With elasticity, this metric value was 4.544598 while 4.905456 without elasticity, showing a difference of 8%.

With these metrics values, it is possible to establish a correspondence between cost and improvement of elasticity to analyze if it is worth to add or remove machines or if it is better to pay penalties. In order to do that, it must be calculated the total spent or saved when adding or removing machines and then compare that total with the cost of paying penalties when no change is made in the cluster. It follows as a suggestion, since cost issues are out of scope of this work.

Metrics results for update operations of Workload A are shown on Table 5.3 and distribution is represented on box plot chart of Figure 5.12. If we compare Figure 5.11 and Figure 5.12, we can see that in the latter 1^{st} percentile is closer to the minimum response time and 3^{rd} quartiles are lower for update operations. This is expected because Cassandra is write-optimized and then the response times of write operations are expected to be lower than the ones for read operations. For *underprov* of update operations our model could also capture an improvement when adding machines while system was

	Update operation	
	with elasticity	without elasticity
Minimum (in μs)	336	351
Maximum (in μs)	31533987	63252175
Average (in μs)	44922.35	53007.87
Median (in μs)	4316	7078
1^{st} percentile (in μs)	426	431
99^{th} percentile (in μs)	223650	253697
1^{st} quartile - Q1 (in μs)	1272	1356
3^{rd} quartile - Q3 (in μs)	23568	42667
# of underprov queries	868136	1496481
# of overprov queries	12980632	10737834
Total of queries	16849140	15630459
Underprov metric	1.393522	1.459517
Overprov metric	19.162614	20.195725
Elasticitydb metric	4.355037	4.582218

Table 5.3: Metrics calculated for Update operations

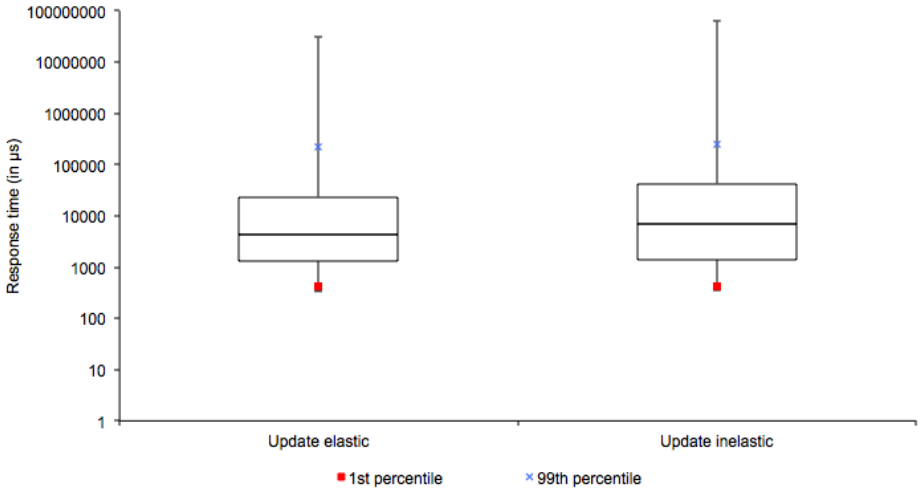

Figure 5.12: Box plot chart showing elastic and inelastic scenario of Update operation - Workload A

overloaded. *underprov* value for elastic scenario was 1.393522 and 1.459517 for inelastic one. Unlike what happened on read operations metrics, *overprov* of update operations is lower in the elastic experiment. Regarding to $elasticity_{db}$, there were an improvement of 9% in the elastic scenario.

5.3.2 Workload E

In workload E, two kinds of operations are performed: scan and insert. However, unlike workload A, number of queries is not evenly split between the two operation types. In this workload, 95% of the queries are scan while 5% are insert. Thus, this workload is read-intensive, while workload A is write-intensive.

Figure 5.13 shows the moments where machines are added or removed. Similarly to workload-A experiments, the time to boostrap Cassandra nodes varied a lot. To add the 1^{st} machine 5min were taken, while 21min to the 2^{nd} and 4min to the last 3^{rd} one. Decommissioning nodes in this workload took from 2min to 4min and the reason is the same explained in Subsection 5.3.1.

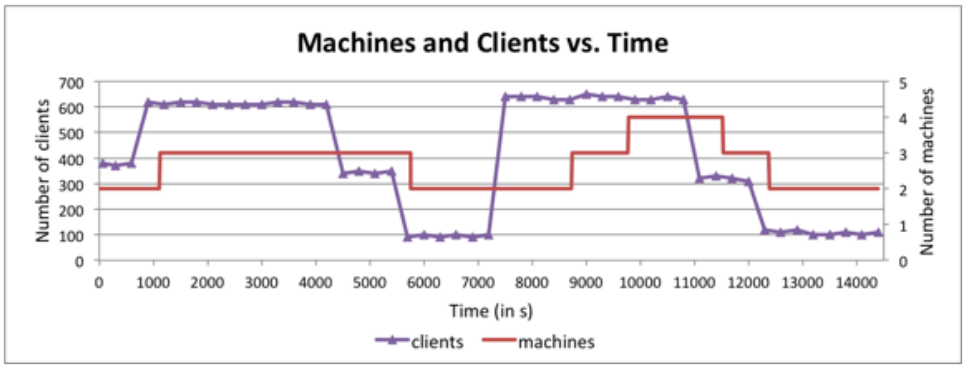

Figure 5.13: Scenario with elasticity

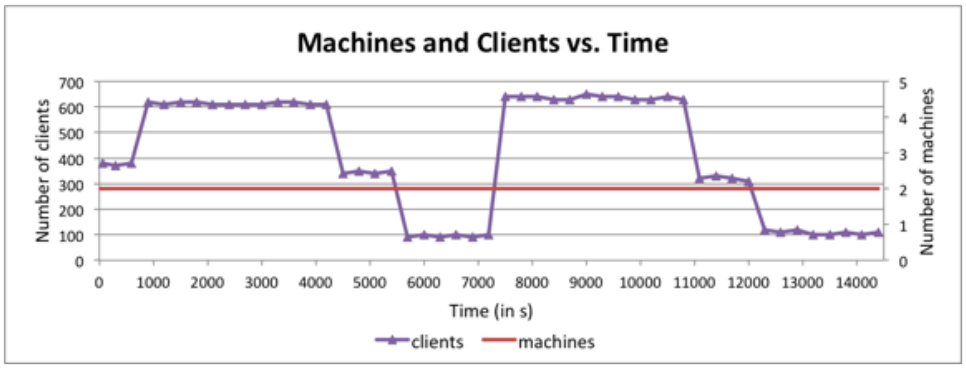

Figure 5.14: Scenario without elasticity

Figures 5.15 and 5.16 show the average response time by second for scan operations. The number of queries performed in workload E is lower than the number of queries on workload A, although the execution time is the same. This happens because scan queries of Workload E take longer than update and read operations of Workload A.

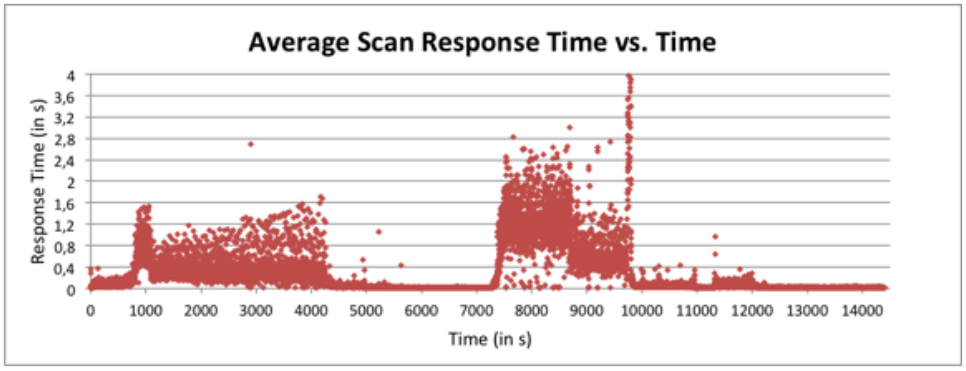

Figure 5.15: Scenario with elasticity

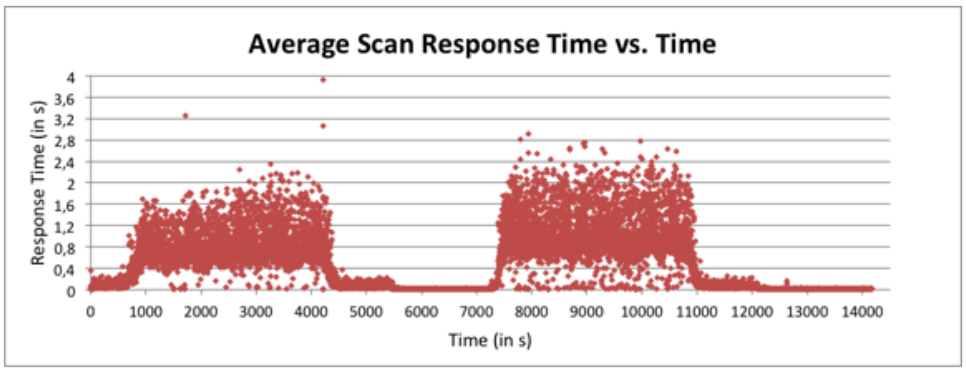

Figure 5.16: Scenario without elasticity

Once the workload duration is preconfigured to 4 hours and workload clients (threads) send queries indefinitely until receiving a stop request, performing quicker queries also means to perform more queries in a fixed timeframe. Figures 5.17 and 5.18 show average response times for insert operations.

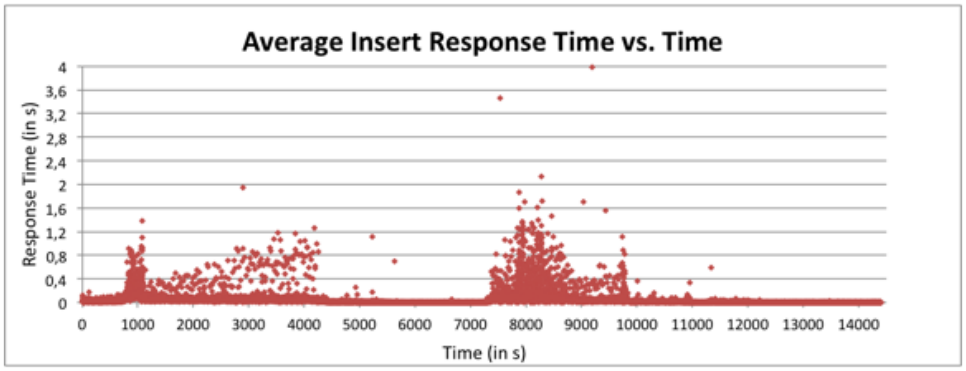

Figure 5.17: Scenario with elasticity

Metrics results for scan operations of Workload A are shown on Table 5.4. In

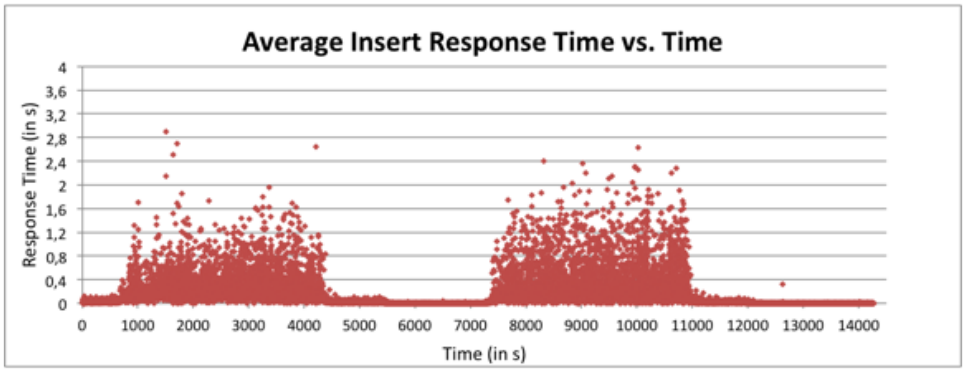

Figure 5.18: Scenario without elasticity

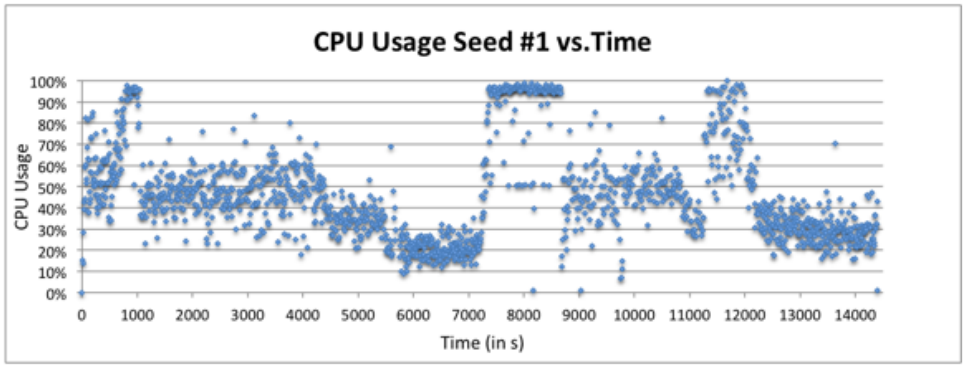

Figure 5.19: Scenario with elasticity

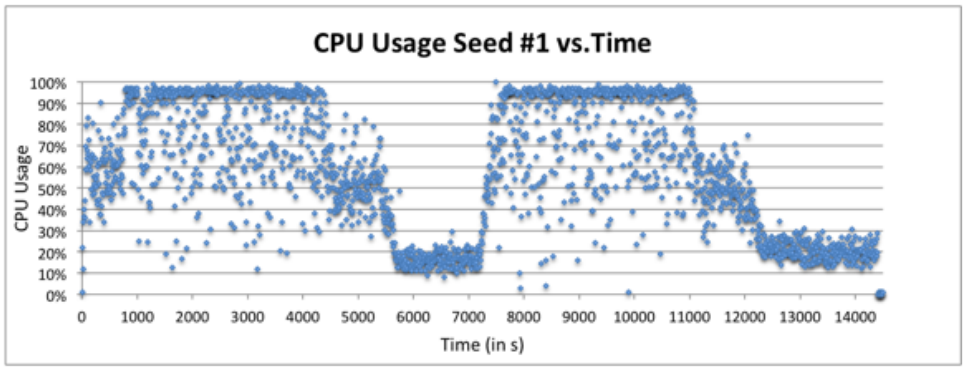

Figure 5.20: Scenario without elasticity

addition, Table 5.4 and Figure 5.23 describes the shape of data distribution. As well as on workload A, the distribution is long-tail.

For *underprov* of scan operations our model could capture an improvement when adding machines while system was overloaded. In this case, the number of violated queries on elastic scenario is 54% of violated ones in the inelastic scenario. *underprov*

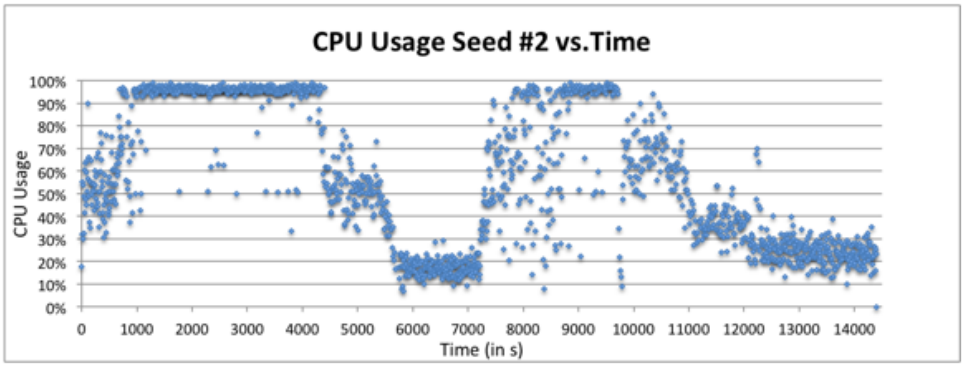

Figure 5.21: Scenario with elasticity

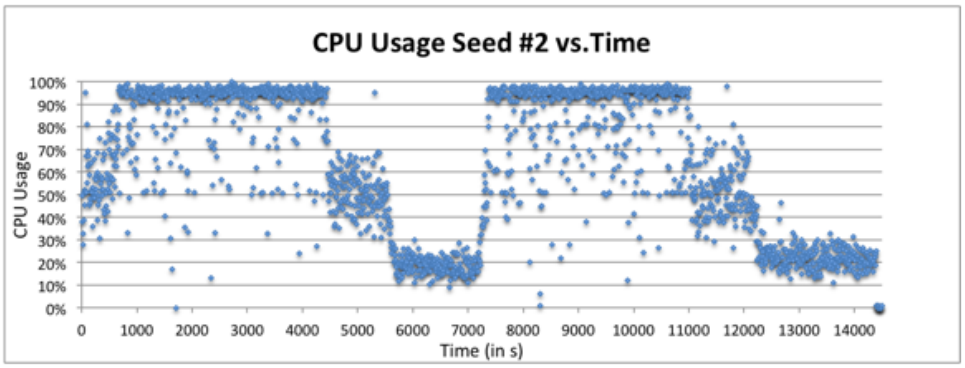

Figure 5.22: Scenario without elasticity

	Scan operation	
	with elasticity	without elasticity
Minimum (in μs)	1545	2340
Maximum (in μs)	72338541	19849025
Average (in μs)	315486.29	601532.67
Median (in μs)	40309	182083
1^{st} percentile (in μs)	3783	4129
99^{th} percentile (in μs)	3675290	3666512
1^{st} quartile - Q1 (in μs)	15582	22806
3^{rd} quartile - Q3 (in μs)	274319	823538
# of underprov queries	528670	962077
# of overprov queries	3064403	1806235
Total of queries	4360855	3590017
Underprov metric	1.988760	2.356267
Overprov metric	11.584427	11.078602
Elasticitydb metric	3.588207	3.809989

Table 5.4: Metrics calculated for Scan operations

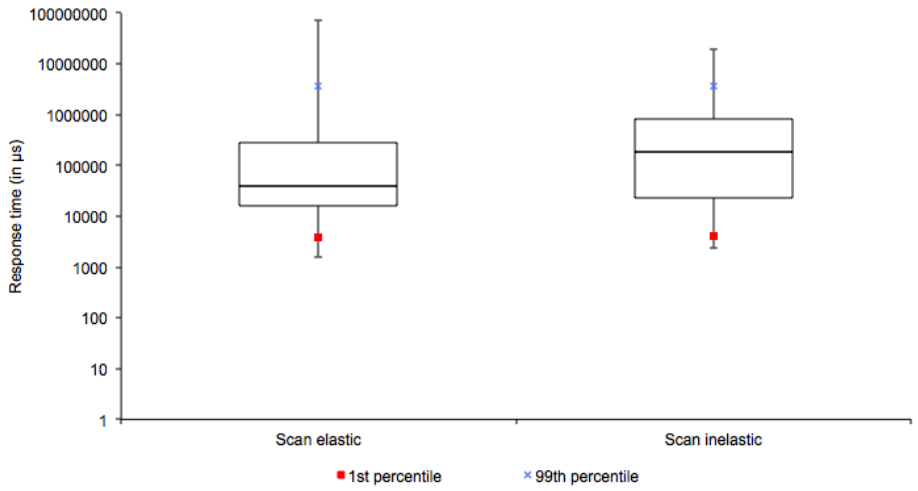

Figure 5.23: Box plot chart showing elastic and inelastic scenario of Scan operation - Workload E

is 1.988760 in elastic scenario, i.e. 84% of this metric in the inelastic scenario. Likewise *overprov* results for read operations, *overprov* of scan operations showed a higher value for the elastic experiment. Some possible reasons for this behavior are the same already explained in the read-operation analysis. $elasticity_{db}$ for the elastic scenario showed an improvement of 9% when compared with the inelastic scenario.

	Insert operation	
	with elasticity	without elasticity
Minimum (in μs)	569	572
Maximum (in μs)	20662772	18503272
Average (in μs)	63304.47	231992.62
Median (in μs)	6908	21294
1^{st} percentile (in μs)	724	691
99^{th} percentile (in μs)	1112349	2396839
1^{st} quartile - Q1 (in μs)	1751	1864
3^{rd} quartile - Q3 (in μs)	37202	181441
# of underprov queries	26070	60345
# of overprov queries	163013	98838
Total of queries	228806	188133
Underprov metric	2.578634	5.792767
Overprov metric	13.459565	16.552327
Elasticitydb metric	4.392123	7.586027

Table 5.5: Metrics calculated for Insert operations

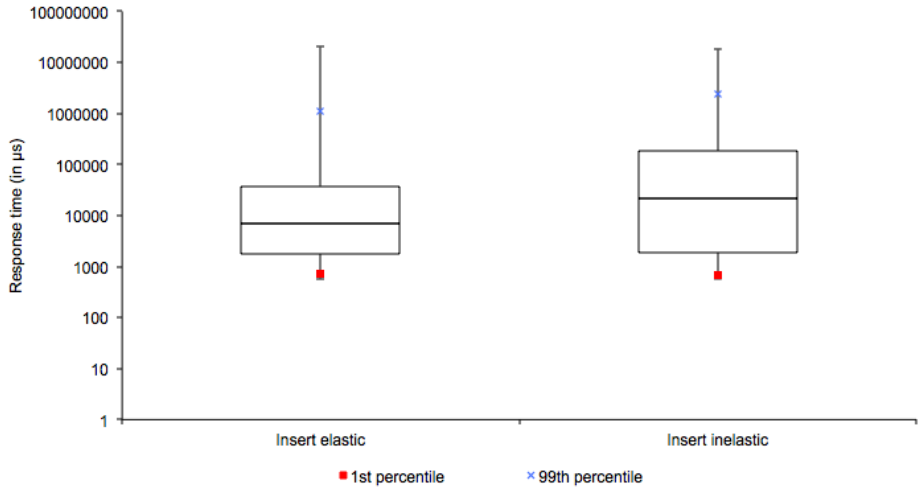

Figure 5.24: Box plot chart showing elastic and inelastic scenario of Insert operation - Workload E

The results gathered for insert operations, shown on Table 5.5 and on Figure 5.24, are similar to values of Update operations of Workload A, regarding to having better values in the elastic scenario. The reason is the way the Cassandra DB Binding of YCSB is implemented and the way Cassandra performs update and insert operations. Calls to update operations of Cassandra DB Binding are merely converted to insert calls. Therefore, in this DB Binding insert and update operations are pretty much the same. Now, suppose that Cassandra DB binding performs different calls for update and for insert operations. Even in this case, these two operation types would be treated pretty much in the same way. For Cassandra, the semantics of Insert and Update are identical. In either case a record is created if none existed before, and updated when it does exist [55].

5.3.3 Varying Parameters of Elasticity Metrics

Our model requires few parameters to calculate elasticity of cloud database systems. In order to better illustrate our model, we show how elasticity metrics vary as one of these parameters changes. To do so, we chose an operation type of an specific workload: Scan operation in elastic scenario of Workload E. After that, we fixed all parameters but one and then recalculated the metrics. Our goal is to help both on understanding the metrics and on figuring suitable parameters values out. Any other operation type or workload could be chosen to illustrate the following metrics results.

Since BenchXtend outputs the response times for individual queries, there is no need to re-execute the workload every time a parameter is changed. We only need to get the output of our tool and input it in a Ruby script [44] that calculates the metrics as our tool does. We varied the following parameters: *underprov* percentile, *overprov* percentile, SLA upper and lower bounds for scan operations and x weight.

As we change only one parameter at each time, we assume the values shown on Table 5.6, except when the parameter is the one being changed. The values on table 5.6 are not meant to be the best values.

Parameter	**Value**
underprov percentile	99^{th}
overprov percentile	1^{st}
SLA upper bound for Scan (in μs)	700000
SLA lower bound for Scan (in μs)	200000
underprov weight (x)	5

Table 5.6: Parameter values for elasticity metrics

Varying *underprov* percentile

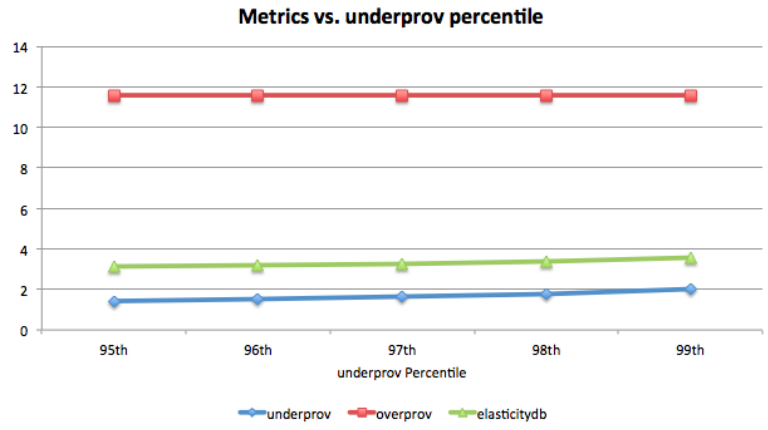

Figure 5.25: Metrics variations when underprov percentile varies

We varied *underprov* percentile from 95^{th} to 99^{th}. Changing the percentile, *underprov* and consequently $elasticity_{db}$ are affected. The lower the *underprov* percentile is, the bigger the number of discarded queries is. That means that queries whose response times are greater than the percentile value are not considered in this metric calculation. Thus, queries that would be potentially very impactful on the *underprov* are not included in the calculation, what reduces the metric value. When percentile increases

from 95^{th} to 99^{th}, *underprov* increases from 1,425088 to 1,988760 (a variation of 39,55%). $elasticity_{db}$ varies less (15,06%), with the same percentile variation, since *overprov* is kept constant. It is worthwhile noting that having low values of *underprov* percentile can generate weird behaviors, like when SLA upper bound is higher than this percentile. In such case, all queries whose response times are higher than SLA upper bound would be discarded from the *underprov* metric calculation and then *underprov* would be equal to zero.

Varying *overprov* percentile

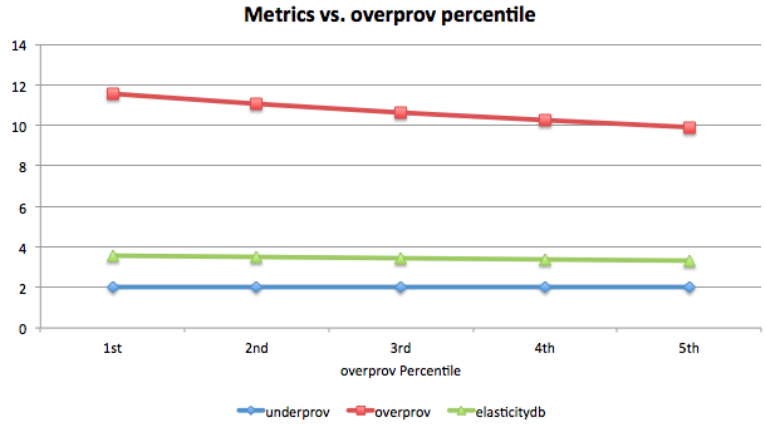

Figure 5.26: Metrics variations when overprov percentile varies

As *overprov* percentile changes, *overprov* and consequently $elasticity_{db}$ varies. The higher this percentile is, the greater the number of queries not included in the *overprov* calculation is. *overprov* varies from 11.585442, with 1^{st} percentile, to 9.909766, with 5^{th} percentile, i.e. 16.90% of variation, as shown on Figure 5.26. $elasticity_{db}$ is directly proportional to *overprov* percentile and varies 8.44% from 1^{st} to 5^{th} percentile.

Varying SLA upper bound

When we increase SLA upper bound keeping SLA lower bound unchanged, we are increasing the acceptance range of queries (see Figure 3.1). Consequently, less queries will be considered under-provisioned and the *underprov* metric is likely to reduce as SLA upper bound increases. Figure 5.27 shows a reduction of *underprov* values as expected. Varying SLA upper bound from 550000μs to 750000μs, *underprov* reduced 15,85% and $elasticity_{db}$ reduced 7,20%.

Varying SLA lower bound

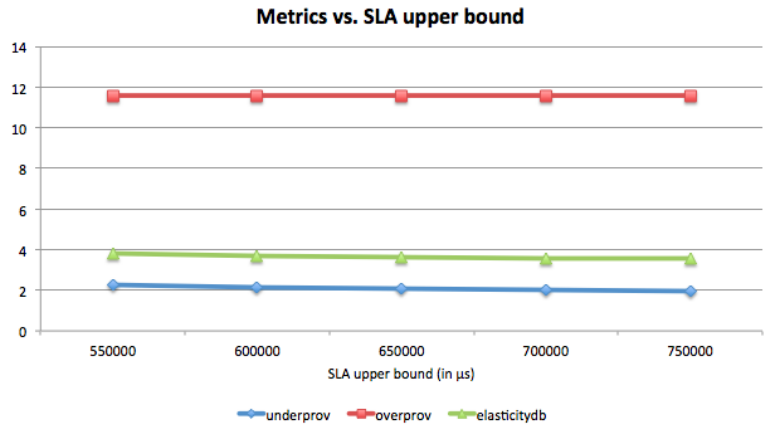

Figure 5.27: Metrics variations when SLA upper bound varies

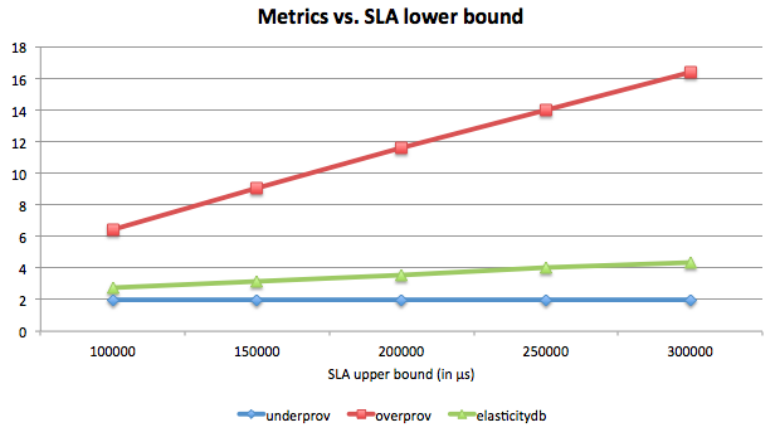

Figure 5.28: Metrics variations when SLA lower bound varies

Unlike what happens when we increase SLA upper bound, when we increase SLA lower bound keeping SLA upper bound unchanged, we are reducing the acceptance range of queries. Thus, more queries will be considered over-provisioned and the *overprov* metric is likely to increase as SLA lower bound increases. Figure 5.28 shows an increase of *overprov* values as expected. Varying SLA lower bound from 100000μs to 300000μs, *overprov* increased considerably from 6.434826 to 16,341697 and $elasticity_{db}$ increased 60,48%.

Varying *underprov* weight

underprov weight is used in the $elasticity_{db}$ metric that is a weight arithmetic mean. As we increase the x weight, the mean trends to converge to *underprov* metric value. This behavior is illustrated on Figure 5.29. $elasticity_{db}$ metric changed from

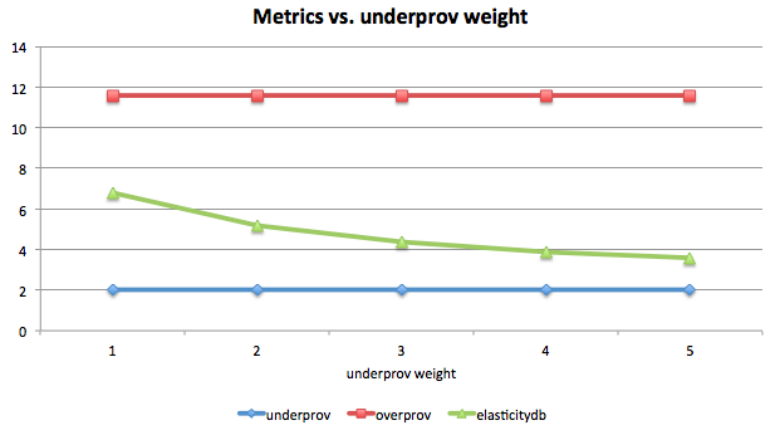

Figure 5.29: Metrics variations when underprov weight varies

6.787101 to 3.588207 as x varied from 1 to 5.

5.3.4 Overall Analysis

In this overall analysis we discuss the results without limiting our considerations to the workloads separately. After that, we present some difficulties faced when configuring the environments and executing the experiments. Finally, we define some rules that must be followed when comparing elasticity of two different database systems.

Metrics Analysis

Analyzing the results, we can see that on Workloads A and E and for all operation types, *underprov* values are lower in the elastic scenario, when compared with the inelastic one. This is the expected behavior and is captured by the our model. However, for *overprov* values, we can see unexpected results for read and scan operations, of workload A and E, respectively. In these cases, inelastic scenarios showed lower values. We have some suspicions to justifiy this behavior. The first possible cause is an inappropriate tuning of Instance Manager to identify the better moment to remove machines. This may be impacting the results but we do not think this is the main reason because we did not face such problems in the results of update and insert operations. Our Instance Manager was used instead of other solutions for a matter of simplicity and since other few solutions we analyzed were not available for Amazon EC2. The second possible cause that comes up to our minds is an overhead of Cassandra operations to bootstrap or decommission nodes due to their time to be performed. The need for such an expensive

operation limits the elasticity of a Cassandra cluster for short-term load variations. We could notice that such Cassandra operations affect negatively the queries being performed while these kinds of operations are being executed. Since we faced such problem only on write operations (update and insert), the different ways Cassandra deals with read and write operations may also have an effect on this behavior. Our last suspicions and the one we think is the most likely to be the root cause of this behavior is the way our *overprov* metric is built.

On *overprov* metric we divide the expected time, which is the SLA lower bound value, by the execution time. Thus, we have an idea on how far from the expected time the over-provisioned query response times are. For scan operations on elastic scenario, for instance, the fraction $\frac{expected_{rt(i)}}{query_{et(i)}}$ reached the maximum of 58.34. This means that there are some queries that are about 58 times lower than the expected time. This idea of having a measure of how far from the expected time the response times actually are is important, but maybe there must have a treatment to deal with some cases or minimize the weight of very quick queries. Besides thinking about the way that fraction is calculated, we should also reconsider whether using an arithmetic mean on *overprov* calculations is suitable or not. Arithmetic mean is sensitive to influence of few extreme observations or outliers. This behavior of mean is likely to be affecting some results.

Problems during Experiments

Initially, we planned to adopt the same variation of clients both for workload A and workload E. We defined a timeline for workload A varying the number of clients from 500 to 2000, as shown on Figure 5.1. Executing the workload A with such timeline was not a problem. However, when executing it with workload E, Cassandra DB binding of BenchXtend started throwing many *TimedOutException*, few minutes after starting the workload. These exceptions stop being thrown only when we killed the benchmark process. After discussing with Cassandra users and specialists, we figured out these exceptions were caused since the Cassandra could not handle the high number of connection requests and queries sent to the database. Checking the metrics gathered by the Monitor component, we could see that CPU and Memory usages were over 95% just before exceptions were thrown and kept at these rates until killing the benchmark process.

Cassandra versions from 1.2.0 present a promissing feature called vnodes [28]. We configured a separate set of Amazon-EC2 instances with Cassandra 1.2.5 in order to test this feature. However, when performing any BenchXtend workload, Cassandra

system used to suddenly stop responsing all the requests. Although we have spent few more days trying to fix this by changing Cassandra parameters, we could not successfully perform workloads on that version. Due to this, we kept using version 1.1.12. Since new versions were released after 1.2.5 was, we believe that the newer versions have fixed the mentioned problem.

Another problem that we faced during our experiments was the performance variations of Amazon EC2 instances. These fluctuations make hard to analyze some results of our tool and our metrics. Even though there are some works that discuss this performance issue of Amazon EC2, to the best of our knowledge, there is no way to isolate or to measure the impact of it.

How to Compare Elasticity of two Different Database Systems

Even though we do not aim to compare the elasticity of different database systems in this work, during our research, we identified two points that are mandatory to be considered when comparing elasticity of database systems:

- Set the same BenchXtend parameters for all systems under test
- Define timelines that equally stress each system under test

In order to measure elasticity of a system with BenchXtend it is necessary to define the parameters used by our elasticity model, listed in Table 4.1, as well as SLA upper and lower bounds, like the ones defined in the Table 5.1. Since these parameters directly affect the measure of elasticity based on our model, it is mandatory to use exactly the same values for all systems under test. All those parameters are independent of database system.

Apart from defining the same BenchXtend parameters, defining the appropriate timeline for each database systems is crucial to actually measuring their elasticities. Usually, traditional benchmarking is performed by defining a workload and by applying the same workload to all systems under test. Although this is suitable to evaluate some database metrics and to provide a basis for fair comparisons, in order to measure elasticity the goal must be to evenly stress the systems. If to stress a system is necessary to have a different workload, then, another workload must be defined, even if it differs from the other workloads.

Suppose that, to perform some experiments, we apply the same workload for database systems A and B. After executing the benchmark and gathering the results, we realize that system A showed lower values for *underprov*, *overprov* and $elasticity_{db}$, when compared to B. One could say that, consequently, A is more elastic than B. If A is more efficient than B with the same amount of resources, A can seem to be more elastic than B. However, if A was not as stressed as B was, we cannot actually evaluate how the system reacts as the load varies. Maybe the system B quickly acted to satisfy a demand as expected to an elastic system, while system A was not stressed enough to check on how it reacts to the demand fluctuation. Thus, a more efficient system is not necessarily a more elastic one and it must be paid attention not to confuse efficiency and elasticity. Our considerations corroborates with [11].

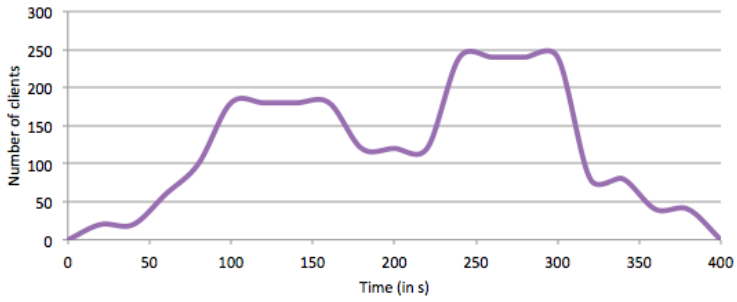

Figure 5.30: Example of timeline with higher peaks

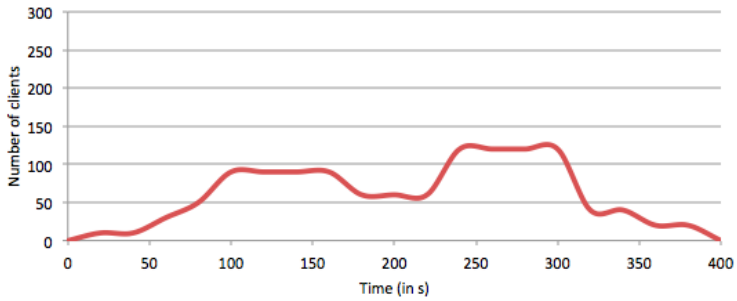

Figure 5.31: Example of timeline with lower peaks

In order to exemplify, we executed workload E on Cassandra and on a MongoDB cluster. We tried to configure MongoDB as much similar as possible to Cassandra cluster, although MongoDB presents some architectural differences, like Config Server, Routers and Shards [56]. From the cloud database system shown on Figure 4.5, we changed only the database system in the pool of resources. BenchXtend and Instance Managers were let the same for both systems. The results of Cassandra for workload E were already

discussed. When executing on MongoDB, the workload E could not stress the system to require the addition of new machines. CPU usage was kept under low rates (most of time between 20% and 40%) during the workload execution, and then, the Decision Taker component of the Instance Manager did not act to increase the cluster.

In conclusion, the timeline must be carefully chosen for each system under test in order to properly stress it. It is worthwhile noting that the shape of the timelines should be similar, to fairly compare how each system deals with, for instance, a spike on demand or two spikes separated by few hours of low demand. Figures 5.30 and 5.31 illustrates two hypothetical examples of similar timelines that could be used to benchmark different database systems.

CHAPTER 6

CONCLUSION AND FUTURE WORKS

In this work, we presented BenchXtend, a benchmark tool to measure elasticity of cloud database systems. This tools extends YCSB tool by adding features to vary the number of clients during workload execution and to calculate elasticity metrics. This variation allows to properly stress database systems under test, which is mandatory to evaluate the elasticity of a system. Due to the lack of a model that describes elasticity of database systems based on quality of service, we also proposed a model with a set of metrics to measure elasticity of cloud database systems. Our model presented three metrics based on SLA and from consumers and providers perspectives. These metrics consider both under-provisioning and over-provisioning scenarios and their values are dimensionless scalars, what allows us to easily compare the results of different database systems.

According to the analysis of our experiments, our tool could properly vary the load during execution in order to stress Cassandra database system. By stressing the system we could validate our metrics and *underprov* correctly captured the variation of elasticity for all evaluated operations in elastic and inelastic scenarios. For *overprov* of read operations our results showed controversial behavior, because inelastic scenario presented better results than experiments on elastic scenario. We presented few possible causes to explain the unexpected results of *overprov*, but an additional analysis must be performed to confirm the root cause of the gathered results. These metrics can now be used to compare elasticity of cloud database systems and to help providers on improving or tuning strategies to add or to remove resources depending on the demand. Even though we have not included in the scope of this work a comparison of elasticity of different cloud database systems, we presented the requirements that must be considered when comparing different systems.

Our experiments were performed over a Cassandra cluster. The time to bootstrap a Cassandra node varied a lot and sometimes took several minutes to finish, although the size of the database was only about 4GB. The need for such bootstrap operations limits the elasticity of a Cassandra cluster for short-term load variations. Our conclusion

is that Cassandra is more suitable for systems that do not require very rapid reactions to attend spikes of demand.

As future work, we identified that improvements could be made in the benchmark tool, in the elasticity model and in the experiments. Regarding to the benchmark tool, we intend to implement a better load balancer for Cassandra, maybe implementing a new Cassandra DB binding with a newer API of Cassandra, instead of the legacy Thrift API used by YCSB. In addition, we intend to alter also BenchXtend to better integrate it with Instance Manager to switch database connection of clients that are connected to machines being removed.

For the elasticity model, we intend to reduce the number of parameters by performing statistical analyses or by empirically defining values that can be used as standard ones. If we reduce the number of parameters, consequently, we reduce the time spent to figure out the suitable values for parameters like SLA bounds and *overprov* weight. We also intend to evaluate other mathematical devices that could be used to replace the use of arithmetic mean in our model. In addition, we want to investigate the influence on the results of the fractions adopted both on *underprov* and *overprov* calculations.

Regarding to the experiments, we intend to execute experiments to evaluate elasticity of multiple database systems like Cassandra, MongoDB and HBase. In order to do so, we firstly intend to define suitable timelines that can stress each database system. For Cassandra, we want to use a newer version with vnodes feature to check if such feature presents improvements on the time to bootstrap or decommission nodes. Moreover, we intend to perform a new suite of experiments in a private cloud to avoid or, at least, minimize the noisy neighbor problem faced on Amazon EC2. In these new experiments, we want to consider a higher number of machines, varying from 2 to 16 machines.

BIBLIOGRAPHY

[1] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD '11*, pages 301–312, New York, NY, USA, 2011.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[3] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, NY, USA, 2007.

[4] Sherif Sakr. Cloud-hosted databases: Technologies, challenges and opportunities. *Cluster Computing*, November 2013.

[5] Umar F. Minhas, Rui Liu, Ashraf Aboulnaga, Kenneth Salem, Jonathan Ng, and Sean Robertson. Elastic scale-out for partition-based database systems. In *SMDB '12, ICDE Workshops*, pages 281–288, Washington, DC, USA, 2012.

[6] Rodrigo F. Almeida. rodrigofelix/YCSB. `https://github.com/rodrigofelix/YCSB/`, 2013.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, pages 143–154, New York, NY, USA, 2010.

[8] Guilherme Galante and Luis Carlos E. de Bona. A survey on cloud computing elasticity. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, UCC '12, pages 263–270, Washington, DC, USA, 2012. IEEE Computer Society.

[9] Peter Mell and Tim Grance. The nist definition of cloud computing. *National Institute of Standards and Technology*, 53(6):50, 2009.

[10] Francisco Perez-Sorrosal, Marta Patiño Martinez, Ricardo Jimenez-Peris, and Bettina Kemme. Elastic si-cache: consistent and scalable caching in multi-tier architectures. *The VLDB Journal*, 20(6):841–865, December 2011.

[11] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in Cloud Computing: What it is, and What it is Not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 2013), San Jose, CA, June 24–28.* usenix, 2013. Preliminary Version.

[12] Sadeka Islam, Kevin Lee, Alan Fekete, and Anna Liu. How a consumer can measure elasticity for cloud platforms. In *ICPE'12 - Second Joint WOSP/SIPEW International Conference on Performance Engineering*, pages 85–96, New York, NY, USA, 2012.

[13] Peter Brezany, Yan Zhangy, Ivan Janciak, Peng Chen, and Sicen Ye. An elastic olap cloud platform. In *Proceedings of the 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, DASC '11, pages 356–363, Washington, DC, USA, 2011. IEEE Computer Society.

[14] Yu Cao, Chun Chen, Fei Guo, Dawei Jiang, Yuting Lin, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. Es2: A cloud data storage system for supporting both oltp and olap. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 291–302, Washington, DC, USA, 2011. IEEE Computer Society.

[15] Flávio R. C. Sousa, Leonardo O. Moreira, Gustavo A. C. Santos, and Javam C. Machado. Quality of service for database in the cloud. In *2nd International Conference on Cloud Computing and Services Science, CLOSER '12*, pages 595–601, 2012.

[16] Yun Chi, Hyun Jin Moon, Hakan Hacigümüş, and Junichi Tatemura. Sla-tree: a framework for efficiently supporting sla-based decisions in cloud computing. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 129–140, New York, NY, USA, 2011. ACM.

[17] Pengcheng Xiong, Yun Chi, Shenghuo Zhu, Hyun Jin Moon, Calton Pu, and Hakan Hacigumus. Intelligent management of virtualized resources for database systems in

cloud environment. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 87–98, Washington, DC, USA, 2011. IEEE Computer Society.

[18] Divyakant Agrawal, Amr El Abbadi, Sudipto Das, and Aaron J. Elmore. Database scalability, elasticity, and autonomy in the cloud. In *Proceedings of the 16th international conference on Database systems for advanced applications - Volume Part I*, DASFAA'11, pages 2–15, Berlin, Heidelberg, 2011. Springer-Verlag.

[19] Junichi Tatemura, Oliver Po, and Hakan Hacgümüş. Microsharding: a declarative approach to support elastic oltp workloads. *SIGOPS Oper. Syst. Rev.*, 46(1):4–11, February 2012.

[20] Raghu Ramakrishnan. Cap and cloud data management. *Computer*, 45(2):43–49, February 2012.

[21] 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.

[22] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, May 2008.

[23] 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. In *In Proceedings of the 7th USENIX*, pages 15–15, Berkeley, CA, USA, 2006.

[24] MongoDB. Mongodb. http://www.mongodb.org/, 2013.

[25] HBase. Apache hbase. http://hbase.apache.org/, 2013.

[26] Cassandra. The apache cassandra project. http://cassandra.apache.org/, 2013.

[27] DataStax. About Internode Communications (Gossip) — DataStax Cassandra 1.1 Documentation . `http://www.datastax.com/docs/1.1/cluster_architecture/gossip`, 2013.

[28] DataStax. Apache Cassandra 1.2 - About virtual nodes. `http://www.datastax.com/documentation/cassandra/1.2/webhelp/index.html#cassandra/features/../architecture/architectureDataDistributeVnodesUsing_c.html#concept_ds_syz_4mf_fk`, 2013.

[29] Brandon Williams. Virtual nodes in Cassandra 1.2. `http://www.datastax.com/dev/blog/virtual-nodes-in-cassandra-1-2`, 2013.

[30] Carlo A. Curino, Djellel E. Difallah, Andrew Pavlo, and Philippe Cudre-Mauroux. Benchmarking oltp/web databases in the cloud: the oltp-bench framework. In *Proceedings of the fourth international workshop on Cloud data management*, CloudDB '12, pages 17–20, New York, NY, USA, 2012. ACM.

[31] Markus Klems, David Bermbach, and René Weinert. A runtime quality measurement framework for cloud database service systems. In *Proceedings of the 8th International Conference on the Quality of Information and Communications Technology*, pages 38–46. IEEE, Conference Publishing Services (CPS), September 2012.

[32] D.M. Shawky and A.F. Ali. Defining a measure of cloud computing elasticity. In *1st International Conference on Systems and Computer Science, ICSCS 2012*, pages 1–5, Lille, FR, 2012.

[33] Thibault Dory, Boris Mejías, Peter Van Roy, and Nam-Luc Tran. Measuring elasticity for cloud databases. In *Proceedings of the The Second International Conference on Cloud Computing, GRIDs, and Virtualization*, pages 2–15, Berlin, Heidelberg, 2011.

[34] Francisco Cruz, Francisco Maia, Miguel Matos, Rui Oliveira, Joao Paulo, Jose Pereira, and Ricardo Vilaca. Met: Workload aware elasticity for nosql. In *EuroSys 2013*, April 2013.

[35] 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*, pages 725–728, New York, NY, USA, 2012.

[36] Ioannis Konstantinou, Evangelos Angelou, Christina Boumpouka, Dimitrios Tsoumakos, and Nectarios Koziris. On the elasticity of nosql databases over cloud management platforms. In *CIKM '11*, pages 2385–2388, New York, NY, USA, 2011.

[37] DataStax. Benchmarking Top NoSQL Databases - A performance comparison for Architects and IT Managers. `http://www.datastax.com/wp-content/uploads/2013/02/WP-Benchmarking-Top-NoSQL-Databases.pdf`, 2013.

[38] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. Ycsb++: benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 9:1–9:14, New York, NY, USA, 2011. ACM.

[39] Sherif Sakr and Anna Liu. Sla-based and consumer-centric dynamic provisioning for cloud databases. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, pages 360–367, Washington, DC, USA, 2012.

[40] ScaleDB. Database-as-a-service (dbaas). http://www.scaledb.com/DBaaS-Database-as-a-Service.php, 2013.

[41] Cloud Security Alliance. Security guidance for critical areas of focus in cloud computing v3.0. https://downloads.cloudsecurityalliance.org/initiatives/guidance/csaguide.v3.0.pdf, 2013.

[42] Carlo Curino, Evan Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational cloud: A database service for the cloud. In *5th Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2011.

[43] Nist Sematech. Engineering statistics handbook. http://www.itl.nist.gov/div898/handbook/prc/section1/prc16.htm, 2013.

[44] Rodrigo F. Almeida. rodrigofelix/benchxtend-monitor. `https://github.com/rodrigofelix/benchxtend-monitor`, 2013.

[45] TPC. TPC-C - Homepage. `http://www.tpc.org/tpcc/`, 2013.

[46] Jia-Lang Seng, S.Bing Yao, and Alan R. Hevner. Requirements-driven database systems benchmark method. *Decision Support Systems*, 38(4):629 – 648, 2005.

[47] DataStax. Can Cassandra Handle Your Cloud App? Ask Netflix : DataStax. `http://www.datastax.com/2013/04/can-cassandra-handle-your-cloud-app-ask-netflix`, 2013.

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

[49] DataStax. Troubleshooting Guide — DataStax Cassandra 1.1 Documentation. `http://www.datastax.com/docs/1.1/troubleshooting/index`, 2013.

[50] DataStax. About Writes in Cassandra. `http://www.datastax.com/docs/1.1/dml/about_writes`, 2013.

[51] DataStax. Managing a Cassandra Cluster — DataStax Cassandra 1.1 Documentation. `http://www.datastax.com/docs/1.1/cluster_management`, 2013.

[52] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proc. VLDB Endow.*, 3(1-2):460–471, September 2010.

[53] Sean Kenneth Barker and Prashant Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, MMSys '10, pages 35–46, New York, NY, USA, 2010. ACM.

[54] cpuboss. Intel xeon e5507 vs e5-2650. http://cpuboss.com/cpus/Intel-Xeon-E5507-vs-Intel-Xeon-E5-2650, 2013.

[55] DataStax. INSERT — DataStax Cassandra 1.1 Documentation. `http://www.datastax.com/docs/1.1/references/cql/INSERT`, 2013.

[56] MongoDB. Sharding introduction - mongodb manual. http://docs.mongodb.org/manual/core/sharding-introduction/, 2013.

[57] Microsoft. Pricing details - SQL Database — Windows Azure. `http://www.windowsazure.com/en-us/pricing/details/sql-database/`, 2013.

[58] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[59] Rodrigo Almeida. Benchxtend: a tool to benchmark and measure elasticity of cloud databases. In *Simpósio Brasileiro de Bancos de Dados - SBBD 2012 - Workshop de Teses e Dissertações*, pages 93–98, 2012.

[60] Guohui Wang and T. S. Eugene Ng. The impact of virtualization on network performance of amazon ec2 data center. In *Proceedings of the 29th conference on Information communications*, INFOCOM'10, pages 1163–1171, Piscataway, NJ, USA, 2010. IEEE Press.