Towards autonomic workload aware NoSQL databases

Francisco Miguel Carvalho Barros da Cruz

Advisor: Prof. Rui Carlos Mendes de Oliveira

November 2016

DECLARAÇÃO

Nome: Francisco Miguel Carvalho Barros da Cruz

Endereço electrónico: fmcruz@di.uminho.pt Telefone: 918892544

Número do Bilhete de Identidade: 13025225

Título Tese: Towards autonomic workload aware NoSQL databases

Orientador: Prof. Rui Carlos Mendes de Oliveira

Ano de conclusão: 2016

Designação do Doutoramento: The MAP-I Doctoral Program Of The Universities of Minho, Aveiro and Porto

Universidade do Minho, 04/11/2016

Assinatura: _____

STATEMENT OF INTEGRITY

I hereby declare having conducted my thesis with integrity. I confirm that I have not used plagiarism or any form of falsification of results in the process of the thesis elaboration.

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

University of Minho, _____

Full name: _____

Signature: _____

Agradecimentos

No culminar desta longa jornada, apercebi-me que o caminho foi mais importante do que o destino. Olhando para trás vejo que os pontos altos desta viagem foram mais do que as pequenas e grandes vitórias que alcancei, foram as pessoas com quem as partilhei, a quem é indispensável deixar neste projecto o meu profundo agradecimento.

Ao meu orientador, Professor Rui Oliveira por me mostrar que as ideias surgem olhando na mesma direcção numa diferente perspectiva. Pela dedicação, atenção e disponibilidade para todos os projectos que temos em comum. Pela capacidade de me inspirar, fazendo assim o meu trabalho parte do seu legado.

A todos os elementos do HASLab, em particular ao Grupo de Sistemas Distribuídos, pelo meio envolvente de partilha e boa disposição que criaram, do qual me orgulho em pertencer. Em particular ao Professor José Orlando Pereira pela abertura, disponibilidade e paciência para a discussão de qualquer temática.

A todos os companheiros que marcaram a sua passagem pelo laboratório e OsSemEstatuto, nomeadamente Ana Nunes, Fábio Coelho, Filipe Campos, Francisco Maia, Francisco Neves, Jácome Cunha, João Tiago, Miguel Borges, Miguel Matos, Nelson Gonçalves, Nuno Carvalho, Nuno Castro, Paula Rodrigues, Paulo Jesus, Pedro Gomes, Ricardo Gonçalves, Ricardo Vilaça, Rui Gonçalves, Rui Ribeiro e Tiago Jorge. Com quem partilhei os melhores momentos de distração, cansaço e diversão sem os quais não teria sido possível chegar onde me encontro hoje.

Não posso deixar de agradecer à minha família, aos meus pais, Francisco Xavier e Maria da Luz, e à minha irmã, Mariana, que me mostraram, desde que me conheço, que cada viagem começa com pequenos passos e que cada um deles só faz sentido se ladeado e apoiado por pessoas especiais.

Quero deixar um agradecimento muito especial à Inês, pelo apoio e dedicação

incondicionais, que por onde quer que vá, se tem tornado parte de quem sou.

Finalmente queria deixar um profundo obrigado aos meus amigos, dos bons momentos e dos menos bons, não só deste últimos anos mas por todos os outros em que perceberam quem eu sou, aceitaram no que me tornei e gentilmente me ajudam a crescer.

Adicionalmente, agradeço também à instituições que apoiaram o trabalho apresentado nesta tese: à Fundação para a Ciência e Tecnologia (FCT), que apoiou este trabalho através da bolsa de doutoramento (SFRH/BD/80111/2011), e ao Departamento de Informática da Universidade do Minho e ao HASLab - High Assurance Software Lab, que ofereceram-me as condições necessárias para realizar a presente tese.

Braga, Fevereiro de 2016
Francisco Cruz

Towards autonomic workload aware NoSQL databases

In order to attain the promises of the *Cloud Computing paradigm*, systems need to transparently adapt to environment changes. NoSQL databases, which are pivotal systems in nowadays cloud infrastructures, exhibit the highly desirable scalability and availability properties. Scalability achieved by these databases is anchored on data independence; there is no clear relationship between data, and atomic inter-node operations are not a concern. Such assumption over data allows a paradigm shift on how to achieve the best performance. Unfortunately, current solutions put the burden on the application's developer to handle and master the specificities of each system that is hindering a broader adoption.

In this dissertation, we tackle the several shortcomings in current implementations of cloud-based NoSQL databases at four different levels. First, we present a cloud-enabled framework for the automatic and heterogeneous reconfiguration of NoSQL databases. This framework enables NoSQL databases to become autonomously elastic while providing a new load balancing component that takes into account data access patterns. Secondly, we propose a novel mechanism to partition data that takes into account the system workload. It estimates, in an autonomous way, a splitting point that leads to optimal load balancing in terms of requests. Then, we develop a mechanism to accurately predict the resource usage of NoSQL databases resorting to an offline trained model. It can accurately estimate in real time the database resource usage for any request distribution only by knowing two parameters: i) cache hit ratio; and ii) incoming throughput. This mechanism is sufficiently simple and generic so it can be used with several databases. Finally, we leverage the work on the resource usage prediction to design and implement a novel load balancer mechanism that maximizes the resource usage across the cluster.

Rumo a bases de dados NoSQL autonomamente adaptáveis à distribuição dos pedidos

De modo a alcançar as promessas do paradigma da *computação na nuvem*, os sistemas têm de ser capazes de adaptar-se às mudanças de uma forma transparente. Os bancos de dados NoSQL que são sistemas cruciais nas infra-estruturas da *nuvem*, possuem as propriedades de escalabilidade e elevada disponibilidade. A escalabilidade está assente na independência de dados; pois não existe uma relação clara entre os mesmos, e operações atómicas que envolvam mais que um nó não são uma preocupação. Tal pressuposto permite uma mudança de paradigma na forma de alcançar o melhor desempenho. Infelizmente, as soluções atuais requerem responsabilidades adicionais a quem desenvolve as aplicações, nomeadamente a necessidade de manipular e dominar as especificidades de cada sistema. Esta situação está a dificultar a adoção do paradigma.

Nesta dissertação, abordamos várias lacunas das atuais implementações de bases de dados NoSQL a quatro diferentes níveis. Primeiro, apresentamos um sistema que permite a reconfiguração automática e heterogénea de bases de dados NoSQL, que permite que essas bases de dados se tornem autonomamente elásticas e simultaneamente balancear a carga tendo em conta os padrões de acesso. Em segundo lugar, propomos um novo mecanismo de particionamento de dados que dado o estado atual do sistema, estima de forma autónoma qual o ponto ideal de divisão baseado nos pedidos. Desenvolvemos, ainda, um mecanismo para prever com precisão o uso de recursos pelas bases de dados NoSQL com base num modelo construído em modo *off-line*. Esse modelo permite estimar com elevada precisão e em tempo real o uso de recursos da base de dados para qualquer distribuíção somente conhecendo dois parâmetros: i) a taxa de acessos com sucesso da *cache* e ii) o desempenho. O mecanismo é suficientemente simples e genérico podendo ser utilizado em várias bases de dados. Finalmente, tirámos partido do trabalho sobre a previsão de uso de recursos para projetar e implementar um novo mecanismo de balanceamento de carga que maximiza o uso de recursos em todo o *cluster*.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The constant technology evolution, wired as the optical fiber, or wireless such as the WiMAX and LTE are turning Internet access more and more ubiquitous, faster, better and cheaper. The proliferation of Internet access allows users to consume services directly provided over the Internet, which results in a change of paradigm for the use of applications and how users communicate, popularizing the paradigm known as *cloud computing*. This paradigm presents itself as the successor to the Grid Computing and Utility Computing, combining the features of resource sharing from the former with the business model established by the latter. In a *cloud computing* environment, the majority of applications, as well as data, do not need to be installed or stored on the user's computer, as they are provided by the *cloud* through dedicated service providers, also known as Cloud Providers (CPs). The Cloud Provider is responsible, for example, for the storage, maintenance and backup of all user information. The user just has to access the platform provided by the CP and only pay for the services she uses and when she needs them - a concept known as pay-as-you-go.

The *cloud* is a complex environment composed of various subsystems that, although different, are expected to exhibit a set of fundamental features: high availability, high performance and elasticity. While high availability and high performance are common goals to all systems, elasticity is specific to the *cloud* environment and closely tied to the pay-as-you-go model. Elasticity can be defined as the ability of a system to grow or shrink its resource consumption according to demand. It is still an open challenge and a topic of a considerable amount of recent research [Owens 2010, Vaquero et al. 2011].

The ability to adjust resource consumption according to demand, favors the pay-as-you-go model and improves resource utilization. In addition, current CPs make their *cloud* platforms available in three main ways [Armbrust et al. 2010], namely:

- Infrastructure-as-a-Service (IaaS): provides virtualized hardware resources such as computing, storage and networking. The resources are allocated on demand and in a pay-per-use fashion. An example of IaaS is Amazon EC2 [EC2] (for computing) and Amazon S3 [S3] (for storage);

- Platform-as-a-Service (PaaS): offers an encapsulation of a development environment abstraction that can be used to develop, deploy and run applications. Examples include the Google App Engine [AppEngine] and Microsoft Azure [Azure];

- Software-as-a-Service (SaaS): features full applications or generic software like databases, which are offered as a service and accessible as a web service or through a web browser. SalesForce.com and the Google Apps like Gmail are some well known instances of this type.

Regardless of the *cloud* platform type, they are shared by multiple customers in a multi-tenant environment. Therefore, optimal resource utilization becomes an even greater concern, since if one customer is using more resources than needed, it may impact the performance of other customer's applications, resulting in poorer overall performance. From a CP perspective, the ability to dynamically optimize resource usage according to the contracted level of service is fundamental to the business model.

Applications provided by the *cloud* imply the access of millions of users to the same application and partly as a result, storage of digital data has reached unprecedented levels[Skillicorn 2002] with the ever increasing demand for information in electronic formats by individuals and organizations, ranging from the traditional storage media for music, photos and movies, to the emergence of massive applications such as social networking platforms.

Relational Database Management System (RDBMS), which have been the norm in data management systems, are not well suited for these environments.

Consequently, the major online players researched alternatives into building extreme large scale storage systems and the result is a great set of different distributed key-value data stores: Dynamo [DeCandia et al. 2007], PNUTS [Cooper et al. 2008], BigTable [Chang et al. 2006], HBase [George 2011], Cassandra [L. and M. 2009], DataDroplets [Vilaça et al. 2010], among others. Such data stores are also known as NoSQL databases due to their lack of SQL query language interface as opposed to RDBMS. NoSQL databases provide high availability and scalability in a distributed environment composed by a set of commodity hardware, avoiding the need to invest in very powerful and expensive servers to host the database. In addition, these data stores automatically provide replication, fail-over, load balancing and data distribution. Nevertheless, when compared to the much mature RDBMS, NoSQL databases have some fundamental limitations that should be taken into account. They provide high scalability at the expense of a more relaxed data consistency model and only provide primitive querying and searching capability. Thus, data abstraction and consistency becomes responsibility of the application developers. In addition, although NoSQL databases can handle elasticity, they are not autonomously elastic: an external entity is required to control when and how to add or remove nodes. As a consequence, many applications cannot be easily ported to *cloud computing* environments, which may result in poor quality of service..

1.1 Problem statement and objectives

One of the major challenges associated with *cloud computing* is to provide data persistency services, which, simultaneously, offer ease of programming, data consistency and elasticity. Although NoSQL databases were designed with the *cloud computing* paradigm in mind, they add an extra degree of complexity due to their different requirements, ranging from primitive querying and searching ability, to a huge number of different tunable parameters that affect the performance and behavior of the database, as well as elasticity management, data balancing and data partitioning. All of these features, therefore, have to be managed by the application developer, which makes it more difficult for non-experts to include NoSQL databases in their application stack. While, at the same time it raises novel challenges and opens space for improving current proposals beyond the

state-of-the-art.

The main objective of this thesis is then to tackle the several shortcomings in current implementations of cloud-based NoSQL databases at three different levels: i) by enabling NoSQL databases to become autonomously elastic with automatic tuning of performance related parameters; while ii) simultaneously providing optimized load balancing and iii) improved data partitioning mechanisms.

1.2 Contributions

Along the dissertation we present four novel contributions. Firstly, we devised a framework called MeT that provides automated elasticity for NoSQL databases. Current approaches to automated elasticity for NoSQL databases look at the different cluster nodes as identical entities. Therefore, elasticity is limited to the decision of adding or removing nodes from the cluster according to demand. Introducing the possibility of having cluster nodes configured heterogeneously proved to allow for better performance and resource usage. An outcome that is only possible when taking the workload into account. We achieve this by leveraging on an existing IaaS system as the basic provider of elasticity. We expose new database engine metrics regarding workload's access patterns, which are constantly monitored along with the IaaS nodes. This information feeds the decision component that then performs online cluster reconfiguration as needed. The prototype is compatible with HBase and OpenStack [OpenStack Foundation] as the underlying IaaS.

As a second contribution, we propose a novel automated mechanism to partition data that takes into account the system workload. The mechanism proposed estimates, in an autonomous way, a splitting point that leads to optimal load balancing in terms of requests. We show that the algorithm is as simple as effective and it is a good complement to MeT. However, it is a generic approach applicable to different NoSQL databases.

The third contribution of this thesis is a mechanism to accurately predict the resource usage of NoSQL databases. We observed that the majority of the NoSQL systems make use of buffer caching mechanisms to improve performance. Moreover, the effectiveness of such mechanisms is directly related to the performance

and, as a consequence, to the resource utilization of the database. This effectiveness can be measured in terms of the hit ratio that the caching mechanism exhibits. The higher the cache hit ratio the more effective the cache mechanism is, and thus more performant is the database. We propose that instead of a specific workload that is characterized by the three common parameters, namely: i) data size; ii) distribution of requests and iii) incoming throughput, a workload can be characterized by the incoming throughput and by its cache hit ratio, as the latter is a reflection of the i) data size and of the ii) distribution of requests. By taking advantage of this relationship we can use the cache hit ratio and the throughput to build a server usage model based on the *uniform* distribution of requests, that can then be used to predict the resource utilization of any workload only by knowing those two parameters. Both input values can be observed in real time or synthesized for request allocation decisions. This novel approach is sufficiently simple and generic so it can be used with several NoSQL, such as HBase and Cassandra, while simultaneously being suitable for other practical applications.

One of such applications is a resource usage load balancer that can be integrated in MeT framework. This is the fourth and final contribution, the new load balancer maximizes the resource usage of every node composing the cluster, which in turn minimizes the number of nodes needed to meet the desired performance.

1.3 Results

The work discussed in this thesis resulted in a number of publications in distinct international conferences:

- Francisco Cruz, Francisco Maia, Miguel Matos, Rui Oliveira, João Paulo, José Pereira, and Ricardo Vilaça. MeT: Workload Aware Elasticity for NoSQL. In *Proceedings of the ACM European Conference on Computer Systems (EUROSYS)*. ACM, 2013

- Francisco Cruz, Francisco Maia, Rui Oliveira and Ricardo Vilaça. Workload-aware table splitting for NoSQL. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC)*, 2014

- Francisco Cruz, Francisco Maia, Miguel Matos, Rui Oliveira, João Paulo, José Pereira, and Ricardo Vilaça. Resource usage prediction in distributed key-value datastores. In *Proceedings of Distributed Applications and Interoperable Systems (DAIS)*, 2016

Also, preliminary work has been published that greatly improved our knowledge of NoSQL databases:

- Ricardo Vilaça, Francisco Cruz and Rui Oliveira. On the expressiveness and trade-offs of large scale tuple stores. In *Proceedings of the On the Move to Meaningful Internet Systems (DOA)*, 2010

- Francisco Cruz, Pedro Gomes, Rui Oliveira and José Pereira. Assessing NoSQL databases for telecom applications, In *Proceedings of the IEEE 13th Conference on Commerce and Enterprise Computing (CEC)*, 2011

Additionally, the result of collaborations paving the way for this thesis or leveraging its research appear in the following publications:

- Ricardo Vilaça, Francisco Cruz, José Pereira and Rui Oliveira. An effective scalable SQL engine for NoSQL databases. In *Proceedings of Distributed Applications and Interoperable Systems (DAIS)*, 2013

- Fabio Coelho, Francisco Cruz, Ricardo Vilaça, José Pereira and Rui Oliveira. pH1: A Transactional Middleware for NoSQL. In *Proceedings of 33th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, 2014

- Ricardo Jimenez-Peris, Marta Patino-Martinez, Bettina Kemme, Ivan Brondino, José Pereira , Ricardo Vilaça, Francisco Cruz, Rui Oliveira and Yousuf Ahmad. CumuloNimbo: A Cloud Scalable Multi-tier SQL Database. In *IEEE Data Engineering Bulletin*, 38(1): 73-83, 2015

1.4 Outline

The rest of the document is structured as follows. We begin by providing some background and context in Chapter 2 specifically on NoSQL databases with a particular emphasis in two databases, namely HBase and Cassandra. We also survey

related work on elasticity of NoSQL databases as well as autonomous data partitioning and on resource usage prediction. Chapter 3 introduces MeT a cloud-enabled framework that can automatically manage, configure and re-configure a NoSQL cluster in a heterogeneous fashion, according to its access patterns. Equipping the underlying NoSQL database with the ability to be autonomously elastic, by the addition or removal of nodes specifically configured to the load they are expected to serve. In Chapter 4 we describe an automated mechanism to partition data that takes into account the system workload. In Chapter 5 we propose a mechanism to predict the resource usage of NoSQL databases based on the cache hit ratio. This novel mechanism is sufficiently simple and generic so it can be used with several NoSQL systems, such as HBase and Cassandra, while simultaneously being suitable for other practical applications. One of such applications is a resource usage load balancer for NoSQL databases that is described in Chapter 6. Finally, Chapter 7 concludes the thesis and discusses possible future work.

Chapter 2

Background

This chapter focuses on providing some context for the work described in subsequent chapters. Considering that our goal is to tackle several shortcomings in current implementations of cloud-based NoSQL databases we center the chapter around them, beginning by providing an overview of the most important instances. We then focus on two of most popular and representative NoSQL databases HBase and Cassandra by laying down all their features and differences in architecture, and the different components that serve as the main motivation for this work. Subsequently, we provide a brief historical overview over elasticity and more specifically on state-of-the-art systems that aim to bring autonomous elasticity to NoSQL databases, which is the foundation for the MeT framework. We also include some background on autonomous data partitioning. And, finally we provide some background on resource usage prediction and its importance to achieve optimized load balancing and thus optimized resource utilization throughout the cluster specially in the context of a *cloud* environment.

2.1 NoSQL databases

With the demand for elastic and scalable distributed databases for managing large volumes of data, major companies like Google, Amazon, Yahoo! and Facebook came up with their own designs of shared-nothing large scale distributed databases (also known as NoSQL databases) respectively: BigTable [Chang et al. 2006], Dynamo [DeCandia et al. 2007], PNUTS [Cooper et al. 2008] and HBase [George 2011], and Cassandra [L. and M. 2009].

Amazon's Dynamo is a highly available key-value store. It is not accessible as a service, but it is used as a building block to some of Amazon's most popular web services such as S3 or EC2.

Designed to be highly scalable, PNUTS is shared by Yahoo!'s multiple applications. It allows for concurrent queries and data updates. In fact, PNUTS makes part of a service called Sherpa, which also encompasses the Yahoo! Message Broker (a topic based publish-subscribe system).

HBase is the open-source implementation of Google's BigTable, which is a distributed database used internally at Google for web indexing, Google Earth, Google Finance and it is also used to support Google's App Engine. Nonetheless, there are some differences between the two systems: BigTable is built on top of the Google File System [Ghemawat et al. 2003] while HBase relies on the Hadoop distributed file system (HDFS) [Apache]; BigTable's uses Google's Chubby [Burrows 2006] as its distributed coordination service, while HBase uses Zookeeper [Hunt et al. 2010].

Initially developed at Facebook to support the social networking application, Cassandra is now an Apache open-source project. It is a distributed database that encompasses concepts from the Dynamo's architecture and the data model from BigTable.

NoSQL databases run in a distributed setting with hundreds or thousands of machines. Enabling distributed processing over this kind of massive-scale storage poses several challenges: problems of data placement and replication, distributed processing and aggregation. Furthermore, dependability becomes, once again, a major challenge in face of massive distributed data management. Due to the highly dynamic membership inherent to these systems, not only global agreement is usually unreachable but also the overhead caused by state update on readmissions can be unbearable. All of the aforementioned NoSQL databases share a similar approach, however they have significant differences on both the data model and on the architecture proposed.

2.1.1 HBase and Cassandra

In the context of this thesis we will now focus on two of the most popular and widely used NoSQL databases, namely: HBase and Cassandra. According to their architecture, NoSQL databases may be categorized in two main types: fully

decentralized and hierarchical. Cassandra is a representative example of a NoSQL database that follows a fully decentralized architecture, while HBase falls in the hierarchical category.

In the fully decentralized type physical nodes are kept organized on a logical ring overlay. Each node maintains complete information about the overlay membership, being therefore able to reach every other node. Each node composing the cluster is identical.

In the hierarchical type, a small set of nodes is responsible for maintaining data partitions and coordinate processing and storage nodes. HBase organizes tuples into *Regions*: horizontal partitions of tuples. HBase is composed of three different types of servers: master, *RegionServers*, and lock servers. Master servers coordinate the *RegionServers* by assigning and mapping *Regions* to them, and redistributing tasks as needed.

When compared to the relational model, the implemented data model in a NoSQL database is usually rather simple. Both Cassandra and HBase are based on BigTable's data model that can be thought of as a multi-dimension sorted map. The *Keyspace* is a namespace for ColumnFamilies, which in turn map rows to a set of columns. In fact, there is a rough correspondence between a ColumnFamily and a table in the relational model, but they differ on the property that within a ColumnFamily each row can have a completely different set of columns. As a result, there is no pre-defined schema so columns can be added dynamically. Within a ColumnFamily columns can be ordered according to their names, using one of the following supported criteria: ASCII, UTF-8, Long, UUID or binary ordering. Actually, Cassandra extends BigTable's data model by adding a higher data structure: SuperColumnFamilies where each of its attribute columns (in this structure named SuperColumns) can have a list of ordinary columns.

Also Contrasting with RDBMS, these databases only provide a simple key-value interface to manipulate data by means of put, get, delete, and scan operations and they do not offer strong consistency criteria. Complex operations like joining and aggregation are not present and data is denormalized.

Despite having different designs, the mentioned NoSQL databases share common components like request routing and processing, storage and use common distributed systems techniques such as partition, replication and failure detection. In the following, it is highlighted the similarities and differences between

each database regarding those components.

Request routing

Due to the multiple nodes used and the partition of data, every time a request is issued to the database, that request has to be routed to the responsible node for that piece of data. In Cassandra any incoming request can be issued to any node in the system. Then, the request is directly routed to the responsible node. HBase uses a client library that caches *regions* locations and therefore clients send the requests directly to the proper *RegionServer*. When the client starts, and thus its cache is empty, the client has to contact the master server and scan a special table called *hbase:meta* in order to know the *RegionServer* that is responsible for the piece of data.

Consistency model

According to the CAP theorem[Gilbert and Lynch 2002] it is impossible to have simultaneously strong consistency and high availability in the presence of network partitions. Consequently, in order to meet the high availability requirements and at the same time cope with network partitions, some NoSQL databases chose to relax the consistency criteria. Cassandra makes use of eventual consistency: a relaxed consistency criteria where the updates are propagated to all replicas asynchronously, which means that stale data can be read and conflicts may occur. On the other hand, HBase chose not to tolerate network partition in order to offer a stronger consistency criteria, thus it supports atomic operations on data stored under a single row key which can be used to perform read-modify-write operations, but yet not transactions across several row keys.

Data partitioning and load balancing

The characterization of the architecture is directly related to the way databases realize data partition, which is a major aspect of these distributed databases. In the fully decentralized based architectures, data partition is done in a fully decentralized manner through consistent hashing [Karger et al. 1997], while in the hierarchical based architectures a small set of nodes is responsible for maintaining the data partitions. As it could be expected, Cassandra by default uses

consistent hashing in order to dynamically assign tuples to the available nodes. However, it can also use an ordered partitioning scheme. In HBase, a *region* split is automatically triggered whenever it reaches a certain threshold in size. When the decision to split is made, the database will split it into two *regions* of roughly the same size. As an alternative, the splitting procedure can also be done manually. By manual we mean that it requires a human manually choosing splitting points. The assignment of *regions* to *RegionServers* resorts to a randomized data placement component. The strategy followed by this component is to evenly distribute the load of the cluster based on the number of *regions*, i.e. ensuring every *RegionServer* has a similar number of *regions*.

Buffer caching

In NoSQL datasbases, caching mechanisms are key for improving the overall performance. HBase has a *block cache* implementing the LRU replacement algorithm [Sleator and Tarjan 1985]. Several key-values are grouped into block of configurable size and these blocks are the ones used in the cache mechanism. The block size within the *block cache* is a parameter but defaults to 64KB. In addition, HBase update requests are written to memory - the *memstore* - before being flushed to disk. When the number of files reach a certain threshold the compaction process starts, which chooses some files and combine them into fewer but larger files. Both the *block cache* and the *memstore* have configurable sizes in terms of the total java heap size allocated to a *RegionServer*. Like HBase, Cassandra uses caching to speed up the performance of each node that composes the cluster. However, its implementation differs from HBase in the sense that instead of being block oriented, is oriented towards the row. Therefore this cache, used for read operations, is called *row cache* and also operates under the LRU replacement algorithm with a configurable size. Finally, Cassandra update requests are also first written to memory - the *memtable* - before being flushed sequentially to disk.

Persistent storage

In a NoSQL database the persistency component ensures that writes are made durable. While Cassandra relies on local disks for persistency, HBase uses a storage service, namely the Hadoop distributed file system (HDFS). Each *region*

is stored as an append-only file in HDFS, whose instances are called *DataNodes*. Usually, *RegionServers* are co-located with *DataNodes* to promote the locality of the data being served by the *RegionServer*.

Replication

Another important and mandatory component of these type of databases is replication, which is used not only to ensure fault-tolerance but also to improve performance of read operations by means of load balancing. In Cassandra, replication is done by the node responsible for data, as determined by consistent hashing, by replicating it to the R-1 clockwise successors - with a replication factor of R. In HBase replication is done only at the storage layer by HDFS, with a configurable replication factor of R.

Failure detection

NoSQL databases usually provide failure detection of the nodes that compose the system. In the Cassandra failure detector, each node locally determines if any other node in the system is up or down using an Φ Accrual Failure Detector[Hayashibara et al. 2004]. In HBase, the master server is responsible for the failure detection. Moreover, when a *RegionServer* fails, the master server informs other *RegionServers*, and the *regions* of the failed *RegionServer* are distributed across the set of available *RegionServers*. It is worth noting, that the same behavior occurs when a *RegionServer* is properly shutdown, that is the *regions* it was serving are distributed across the set of available *RegionServers*.

Metrics

HBase and Cassandra export several important metrics related to the database's performance through JMX and their web-interface, for instance: the total number of read, write and scan requests, the number of requests per second, and the cache hit ratio. It is also easy to implement and export new metrics, which is important for both system administrators and management tools like the ones described in this work.

Configuration and homogeneity

NoSQL databases often require extensive fine tuning. Both Cassandra and HBase have many configuration parameters that have a great impact on the performance of the database. One of such parameters is the buffer caching size and the *memstore/memtable* size that allow one to give more privilege to read or write operations, in a continuous fashion. Other parameters allow to adjust the behavior of the garbage collector, the write buffer sizes or the number of threads available to answer incoming requests In addition, HBase and Cassandra allow for the use of compression algorithms that greatly reduce the disk I/O and network traffic between the servers and the persistent storage layer. These configuration tasks are typically manual and dependent on the administrator's expertise. Usually, the system administrator will analyze the expected workloads and homogeneously configure the cluster nodes to cope with the expected load with best performance. Such a configuration takes into account the overall cluster performance and each node in the cluster is configured identically. In this thesis, particularly in Chapter 3, we show that by configuring the cluster heterogeneously the performance can be greatly improved. However, cluster management becomes even more complex, thus an automated management tool becomes mandatory.

2.2 Elasticity

According to NIST's definition [Mell and Grance 2011], elasticity can be defined as follows: "Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time." In addition, in order to achieve elasticity and the illusion of infinite capacity available on demand autonomous allocation and management is required [Armbrust et al. 2010]. As a result, there is a significant amount of research work related with dynamic scale of Cloud applications. A good range of this research work is present in [Vaquero et al. 2011] where many of the current state-of-the-art efforts towards an elastic Cloud are referred.

However, in this thesis we focus on automated elasticity for NoSQL databases. In this regard, there are some works worth mentioning. In [Lim et al. 2010]

and [Trushkowsky et al. 2011b], two systems are presented that allow automated control of an elastic storage system, a distributed file system and a custom storage system, respectively. These works, to the best of our knowledge, represent the first attempts on designing true elastic storage systems. The idea behind these systems is having a control system that gathers information about workloads (request latency, utilization, response time, etc.) and decides whether to start or stop a computing instance. In order the determine which servers are overloaded/underloaded, the system from [Lim et al. 2010] measures CPU utilization while the SCADS director [Trushkowsky et al. 2011b] uses a steady-state performance model to predict whether a server can handle a particular workload, without violating a given latency threshold, according to the workload rate of get and put operations.

Elasticity of NoSQL databases has also been subject to analysis. In emphtiramola work [Konstantinou et al. 2012], three different NoSQL databases (HBase, Cassandra and Riak) are tested in order to assess their elastic capabilities. The paper presents extensive experiments that measure the cost of adding or removing nodes from those NoSQL systems. It is important to notice, however, that this control system is restricted to operations such as add or remove database instances. Data distribution is left as a database responsibility and all instances are considered equal. Furthermore, *tiramola* is oblivious to workload information or any database metric. It relies solely on CPU usage, memory consumption and other system-level metrics for its decision model. Similar behavior is obtained by the use of Amazon's Cloud Watch [CloudWatch] together with Amazon's Auto Scaling [Scaling]. The Amazon Cloud Watch service gathers system metrics while the Auto Scaling allows a user to define rules based on such metrics. These rules define what action to take (add or remove nodes) when certain metric values reach some thresholds.

In this work we also use several systems metrics (CPU utilization, I/O wait and memory usage) that are critical for a storage system and highly impacts server utilization's estimation. However, on top of that we also use database specific performance metrics and workload information. In addition, our framework applies different heterogeneous configurations, a clear departure from previous approaches.

In fact, with respect to heterogeneous configuration of computational in-

stances there is some related research work. In [Soror et al. 2008] the authors propose a system to autonomously change virtual machine configurations in order to adjust how resources are allocated. This allows for a certain virtual machine to be granted more resources if it has higher demand. In this particular case, the idea was applied to virtual machines running relational database management systems. For instance, a certain database management system with an heavy workload would be given more memory, thus boosting performance without impacting other lighter databases, and improving the overall performance. If the same resources would be given to every virtual machine, resources would be wasted and the system would perform below its actual capabilities. The idea of heterogeneous configuration of a pool of computational instances is similar to the one we present in Chapter 3. It differs in the fact that we are dealing at the application level, rather than at the level of the virtual machine controller.

2.3 Autonomous data partitioning

Autonomous data partitioning has been a subject of research work in the area of distributed relational databases [Curino et al. 2010, Tatarowicz et al. 2012, Pavlo et al. 2012]. In these works the main goal is to avoid multi-table queries, and thus to avoid distributed transactions. In other words, the objective of autonomous data partitioning in distributed relational databases is to prevent a transaction of including more than one database node, due to the complexity and performance penalty of distributed atomic commitment protocols.

On the other hand, in NoSQL systems distributed transactions are not a concern so the main objective is always performance and optimized resource utilization. Yahoo! Cloud Datastore Load Balancer [Klems et al. 2012] of which the NoSQL database PNUTS is part of, offers a splitting mechanism. In this system data partitioning is performed in two situations. Based on the table size or on its access load. However, even though this system takes into account table load, it does not use such information to decide the splitting point. On the contrary, the information is only used to determine if a table is a hotspot and if so it should be split. In this case, the Yahoo! Cloud Datastore Load Balancer splits data into two partitions with similar size.

In this work, we propose to include, as a database metric, a suggestion of a

splitting point that has the following guarantees: if the table is split in that point, according to past request patterns, the splitting will result in two equal halves in terms of requests and not in terms of data size. The algorithm described in Chapter 4 uses an online median estimation to estimate the splitting point based on the past requests. This type of approach has never been applied to NoSQL databases.

2.4 Resource usage prediction

Predicting how a system will behave is very important. Critical decisions on resource allocation, systems configuration or even choice of technology typically require extensive testing, which is still not enough to actually predict the behavior of the system in a real deployment. Moreover, the outcome of these decisions has a direct impact on system cost and effectiveness. As a result, there has been a considerable amount of research on prediction resource usage of generic systems such as, virtual machines [Wood et al. 2008] [Sudevalayam and Kulkarni 2011]. But the more generic and broad the system, more complex models are needed, as it must take into account many parameters.

As mentioned in the literature, in order to obtain accurate models with fewer variables, it is key to focus on specific applications [Jennings and Stadler 2014]. Therefore, different applications have different requirements which means they may need dissimilar approaches. This is the case of performance prediction for relational databases focused on online transaction processing (OLTP) [Mozafari et al. 2013b] [Mozafari et al. 2013a] as opposed to performance prediction of NoSQL databases. The different assumptions from relational databases significantly change the required approach to accurately predict the performance of NoSQL databases. Namely, relational database prediction mechanisms must cope with a large number of concurrent and lock-prone transactions and need different models for predicting resources such as CPU, RAM, Disk I/O and database locks. While as explained and demonstrated in this work, NoSQL databases performance can be predicted with a model based on a single resource as the characteristics of these two database environments are intrinsically different. Moreover, a single resource model is also not achievable for research work that predicts the performance of SQL queries because they need to build models for each database

operator (e.g., Sort, Merge Join), which are not present in NoSQL databases [Li et al. 2012]. This is also the case of DBSeer [Mozafari et al. 2013b] that uses query logs from relational databases to build distinct linear models to predict the RAM, CPU, network and disk resources of relational databases.

Regarding the techniques used to predict systems' performance, machine learning and analytical modeling are the most commonly used. These can be used exclusively or in combination, by resorting to time-series analysis [Khan et al. 2012] [Gong et al. 2010], regression models [Desnoyers et al. 2012] [Zhang et al. 2007], and clustering [Singh et al. 2010]. These approaches require lengthy training phases to estimate accurately different workload distributions. It is however possible to reduce the duration of this initial phase by using a less accurate model and then refine it, in runtime, with other machine learning algorithms [Didona et al. 2015].

One of the most important applications of performance prediction for NoSQL databases is related to resource allocation and load balancing in Cloud Computing. Systems like MeT and [Konstantinou et al. 2012] [Klems et al. 2012] [Wang et al. 2012] try to overcome this problem by looking at runtime performance metrics and deciding how to balance the load across machines. Some just add or remove machines when needed, while others are also concerned with data and request distribution, and distinct priorities for requests. Although, some of these systems use statistical models to take better decisions, they make use of iterative algorithms to reach optimal configurations. As a distinct approach, the SCADS director [Trushkowsky et al. 2011a] uses a non-iterative steady-state performance model to predict whether a server can handle a particular workload, without violating a given latency threshold. However, SCADS is designed to keep data only in memory and, as we show in this work, when data fits in memory the resource usage is only affected by the incoming throughput and not by the workload distribution.

As shown in Chapter 5 we are able to predict the performance of NoSQL databases by resorting to a single model. This model is based on the observation that the cache hit ratio has a great impact on the system's performance, and by correlating it with the incoming throughput it is possible to accurately predict the performance of NoSQL databases. As other approaches, it needs offline training, but it does not require system traces or runtime mechanisms to improve the

precision of the estimation.

Chapter 3

Workload Aware Elasticity

In this chapter we focus on the elasticity of NoSQL databases. These databases have been designed to take advantage of large resource pools and provide high availability and high performance. Moreover, they were designed to cope with resource availability changes. For instance, it is possible to add or remove database nodes from the cluster and to have the database handle such change transparently. Even though NoSQL databases can handle elasticity, they are not autonomously elastic: an external entity is required to control when and how to add or remove nodes.

Ideally, nodes would be added to the cluster when it is under heavy load, in order to maintain service levels, and removed in the opposite case, to reduce costs. Simply adding and removing nodes is insufficient. In fact, current approaches consider that all nodes of a NoSQL cluster share identical, and thus homogeneous configurations. But in practice, different applications have different access patterns, which may even change over time. In addition, NoSQL databases assume data partitioning, meaning that even within an application there may exist data hotspots.

As our experiments show, fine tuning the available parameters of a NoSQL database on a per node basis, significantly boosts overall performance, specially when considering the workload characteristics. Consequently, the heterogeneity of data access patterns should be taken into account to optimize the use of available resources.

In the following, we present the design and implementation of MeT, an elastic system that not only adds and removes nodes, but also heterogeneously recon-

figures them according to the observed workloads. We achieve this by leveraging on an existing IaaS system as the basic provider of elasticity. We expose new database engine metrics regarding workload's access patterns, which are constantly monitored along with the IaaS nodes. This information feeds our decision component that then performs online cluster reconfiguration as needed.

3.1 Heterogeneity

In NoSQL databases, data is distributed across the cluster, thus each node is responsible for a subset of data. In clear contrast to relational databases (both single node and distributed), in NoSQL databases the co-location of data partitions in the same node, which are usually queried together, is no longer needed because:

- there is no clear relationship between data from different entities, and data is de-normalized;
- computation is done on the client side, for instance queries joining two data partitions do not take advantage of data co-location;
- NoSQL databases do not provide atomic multi-item operations, thus atomic inter-node operations are not a concern.

The fact that data is fairly unrelated and can be highly partitioned across the NoSQL cluster, allows for a rebalancing of the cluster to maximize performance without further concerns, such as data locality for join operations. In order to achieve it, NoSQL databases often require extensive fine tuning. These configuration tasks are typically manual and dependent on the administrator's expertise. Usually, the system administrator will analyze the expected workloads and homogeneously configure the cluster nodes to cope with the expected load with best performance. Such a configuration takes into account the overall cluster performance and each node in the cluster is configured identically. However, different applications have different data access patterns and even within the same application there may exist data partitions that are hotspots while others are seldom accessed.

The heterogeneity in access patterns should therefore be taken into account during distribution and data partitioning. Moreover, regardless of the application they refer to, data partitions with similar access patterns should be placed in the same physical nodes configured specifically and exclusively to serve them. The heterogeneity in access patterns leads to a heterogeneous cluster, i.e. with different node configurations, optimized to achieve better performance under the expected workloads. For instance, in HBase by increasing the *block cache size* (see Chapter 2) we can have one *RegionServer* optimized for read operations, and thus assign read intensive data partitions (or *Regions*) to that *RegionServer*. For clarity of presentation, we can refer to nodes as *RegionServers* or data partitions as *Regions*. This difference in nomenclature depends on whether we are referring to our algorithms that are independent of the implementation, or whether we are referring our prototype, which is based on HBase.

In the following we set up an experiment that validates our intuition.

3.1.1 Workload description

We evaluated HBase in a multi-tenant environment using YCSB [Cooper et al. 2010] as a workload generator configured with different, but simultaneous workloads. The reason to use different workloads simultaneously, is to simulate a multi-tenant setting as expected in a Cloud environment. YCSB provides six pre-configured workloads that simulate different application scenarios. In order to achieve a overall read/write ratio of approximately 1.9:1 [Chen et al. 2010], we modified the configuration parameters of two workloads, namely of $WorkloadB$ and $WorkloadD$. We used the following workloads:

$WorkloadA$**:** readProportion=50%; updateProportion=50%; Application scenario: session store recording recent actions;

$WorkloadB$**:** updateProportion=100%; Application scenario: stocks management;

$WorkloadC$**:** readProportion=100%; Application scenario: user profile cache, where profiles are constructed elsewhere (e.g., Hadoop);

$WorkloadD$**:** readProportion=5%; insertProportion=95%; Application scenario: logging/history;

WorkloadE: scanProportion=95%; insertProportion=5%; Application scenario: threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id);

WorkloadF: readProportion=50%; readmodifywriteProportion=50%; Application scenario: user database, where user records are read and modified by the user or to record user activity.

All workloads were initially populated with 1,000,000 records, except *WorkloadD*. This workload simulates a logging/history application that produces a very fast growing log, thus it was initially populated with 100,000 records. Overall, the cluster starts with around 7GB of data and during a 30 minute run it grows, on average 6GB.

With the exception of *WorkloadD* with only one data partition, each of the remaining workloads has four data partitions (*Regions* in HBase) of the same size. The keys were drawn from YCSB *hotspot* distribution, with 50% of the requests accessing a subset of keys that account for 40% of the key space. In terms of the load distribution on each data partition, it means that one partition is a hotspot (34% of the requests), other partition has an intermediate load request (26%), and the remaining two have few but evenly distributed requests (20% of the requests each).

3.1.2 Experimental setting

In all experiments, one node acts as master for both HBase and HDFS, and it also holds a Zookeeper instance running in standalone mode. Our HBase cluster was composed of 5 *RegionServers*, each configured with a heap of 3 GB, and 5 *DataNodes*. It is noteworthy that the *RegionServers* were co-located with the *DataNodes* with a replication factor of 2.

We used two other nodes to run the YCSB workload generators: *WorkloadA*, *WorkloadB* and *WorkloadC* in one node, *WorkloadD*, *WorkloadE* and *WorkloadF* on the other. All workloads were configured to run for 30 minutes with a ramp-up time of 2 minutes. In addition, all workloads were run with 50 threads each except for *WorkloadD* with 5 threads. Likewise, there were no limitations imposed on the throughput of each workload except for *WorkloadD* with a target throughput of 1500 operations per second. We have imposed these limits to *WorkloadD* so

that all scenarios had identical conditions and were not, therefore, overly influenced by a too rapid growth of data. In our first experiments data grew so fast that far exceeded the capacity of our 5 *RegionServer* cluster, which would then negatively impact the performance of other workloads (multi-tenancy). This behavior was observed especially in the $Manual-Homogeneous$ strategy explained below.

All nodes used for these experiments have an Intel i3 CPU at 3.1GHz, with 4GB of memory and a local 7200 RPM SATA disk, and are interconnected by a switched Gigabit local area network.

3.1.3 Placement and configuration strategies

We defined three different strategies representative of different data placement and node configurations, namely: $Random-Homogeneous$, $Manual-Homogeneous$ and $Manual-Heterogeneous$.

Random-Homogeneous: This strategy represents the regular behavior of HBase with a manual, homogeneous configuration of nodes and using the out-of-the-box randomized data placement component that evenly distributes data partitions across all cluster nodes. Because it is random, it assumes uniformity on the number of requests per data partition. Besides the necessary optimization of the default configuration parameters of HBase, we also configured the two parameters that allocate a percentage of the available memory for read and write operations (*block cache size* and *memstore size*, respectively; see Chapter 2). We adopted a direct mapping between these two parameters and the overall read/write ratio. That is, we assigned 60% of memory to the *block cache size* for read operations and, 40% to *memstore size* for write operations.

Manual-Homogeneous: In this strategy, we manually balanced data, so hot data partitions would be as dispersed as possible across all nodes. Furthermore, since configurations are homogeneous data partitions were distributed so that the number of read/write requests would be evenly balanced across all nodes. In order to do this, we conducted an exhaustive search to find the best distribution. That meant trying out all possible combinations of data partitions to nodes that balanced the number of read/write requests across all nodes. We evaluated 15

possible distributions and we chose the one that showed better throughput.

Note that this strategy represents one possible distribution that $Random - Homogeneous$ could achieve. The configuration parameters are the same as in $Random - Homogeneous$, so any performance improvement obtained is solely due to the data placement.

Manual-Heterogeneous: In order to take advantage of heterogeneity in access patterns, this strategy comprises manual data placement and heterogeneous node configuration. The objective of this strategy is to cluster data partitions with similar access patterns. In addition, each node is specifically configured according to the type of load it is expected to handle.

The first step to implement this strategy was to observe the workloads described earlier, in order to understand if and how we could cluster them according to their access patterns. Just by looking at the distribution of requests for each workload, one can easily conclude that $WorkloadA$ and $WorkloadF$ have a mix of read/write operations; $WorkloadC$ produces only read operations; $WorkloadE$ is mainly composed of scan operations; while $WorkloadB$ and $WorkloadD$ generate almost only write operations. These observations lead us to our first conclusion: we can aggregate the workloads into four main groups according to their access patterns, namely *Read/Write* mix, *Read*, *Scan* and *Write*.

The next step is related to the mapping of the data partitions to the *RegionServers* available. Intuitively, the number of *RegionServers* to assign each group should be proportional to the number of data partitions it contains. For instance, if we have a *Read* group containing 20 data partitions and a *Write* group containing only 5 data partitions, it is clear the number of *RegionServers* to assign to the *Read* group should be higher than to the *Write* group. Our experiments confirmed this intuition. Consequently, in the current context we used the following distribution: each of the groups considered were assigned one *RegionServer*, except for the *Read/Write* group. In fact, this group was assigned two *RegionServers*, because it contained 8 data partitions as opposed to the 4 or 5 data partitions of the other groups.

Once we have the mapping of groups to *RegionServers*, we distribute data partitions following an approach similar to $Manual - Homogeneous$. In other words, for the data partitions belonging to the *Read/Write* group we balanced data so each of the two *RegionServers* had a similar load (i.e. similar number

of requests). Once more, we resorted to an exhaustive search that culminated with the hotspots of each workload being in different *RegionServers*, and with the same number of data partitions in each *RegionServer* (i.e. 4 data partitions in each one).

After the data placement stage was completed, we then manually configured each *RegionServer* taking into account the load they were expected to handle. For instance, all data partitions belonging to $WorkloadE$ were assigned to a single node with tailored configuration, namely increased *block size* (better for sequential reads) and almost all available memory set for a read workload with only marginal space for writes. On the contrary, the *RegionServer* of $WorkloadB$ and $WorkloadD$ was configured for a write workload.

3.1.4 Results

Figure 5.4 shows the throughput for all workloads under the different HBase strategies detailed above. Each bar in the plot represents a specific observation in the cumulative distributed function (CDF) of the results, for instance the medium shade of gray (50^{th} percentile) indicates half of the observations were below that value and the other half above. All presented results are the consequence of 5 independent runs.

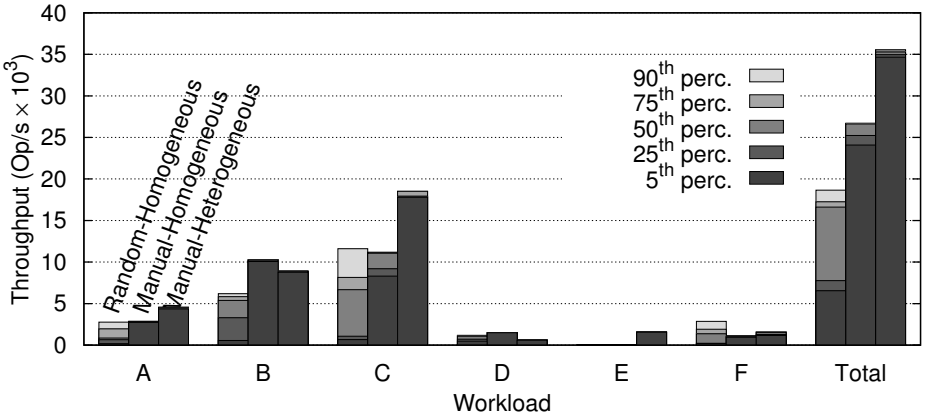


Figure 3.1: Manual strategies results.

It is clear that the $Manual - *$ strategies impact positively the overall cluster performance. While the $Manual - *$ strategies improve to some extent the throughput of $WorkloadA$, $WorkloadB$ and $WorkloadE$, most of the observed

improvement is due to the performance of $WorkloadC$.

The variance observed in the $Random - Homogeneous$ strategy, both in each workload individually and in the total throughput is very high due to the randomness of the data placement component. As it is possible to observe, there was one run whose total throughput was close to $Manual - Homogeneous$'s result, while in another run the total throughput is almost half of the mean. This first comparison confirms that a random data placement, when the distribution of requests is not uniform, may lead to very distinct results. As such, we need to carefully distribute data partitions across the cluster when dealing with a non-uniform distribution of requests. Otherwise, the performance of the cluster is left to chance.

Thereby, with a different strategy on data placement and, accordingly, configuring the nodes for the expected load the results achieved by the $Manual - Heterogeneous$ strategy outperforms the two other strategies. As opposed to the $Manual-Homogeneous$, $Manual-Heterogeneous$ strategy improves each workload in relation to the other two strategies, except marginally for $WorkloadD$. At first glance it may seem that $WorkloadF$'s performance is better under the $Random-Homogeneous$ strategy. This is not true, since on average $WorkloadF$'s performance for the $Random - Homogeneous$ strategy is somewhat lower than under $Manual - Heterogeneous$. Nonetheless, due to the randomness of the data placement component there was at least one run (90th percentile) whose throughput was higher than $Manual - Heterogeneous$ strategy.

Regarding the total throughput, $Manual-Heterogeneous$ more than doubles the result achieved by strategy $Random - Homogeneous$, and in relation to $Manual-Homogeneous$ it improves the result by 35% on average. It is important to stress that for $WorkloadE$ (majority of scan operations) the improvement is remarkable: from around 100 scans per second, to around 1350 scans per second.

3.1.5 Analysis

From the analysis of these results it is possible to see that a heterogeneous HBase cluster can outperform the default configuration. Even when using a judicious data placement, but still with homogeneous nodes, the results are worse than the $Manual - Heterogeneous$. Specifically, NoSQL nodes should not be treated as homogeneous entities because it often results in a skewed load on cluster nodes

leading to both poor resource usage, due to idle nodes, and degraded performance, due to overloaded nodes. These observations motivate our belief that it is not sufficient to simply add or remove HBase nodes in order to have an effective elastic database. Instead, it is necessary to take into account the database workload and adapt the cluster accordingly. Unfortunately, combining heterogeneous node configurations with resource allocation and data placement is a difficult and error prone task, thus should be automated.

Next, we detail the design and implementation of a mechanism that is able to autonomously achieve performance results similar to the heterogeneous configuration and manual data placement, without human intervention.

3.2 MeT Framework

The heterogeneous configuration of a HBase cluster has proven to achieve much better performance than the alternatives. The downside being it greatly increases the complexity of cluster management. In fact, if the number of nodes and data partitions increases to the magnitude of hundreds or thousands, the manual heterogenous configuration of a cluster is impracticable.

As a result, we developed MeT: a cloud-enabled framework that can automatically manage, configure and re-configure a cluster in a heterogeneous fashion, according to its access patterns. Furthermore, MeT equips the underlying NoSQL database with the ability to be autonomously elastic, by the addition or removal of nodes specifically configured to the load they are expected to serve.

Figure 3.2 depicts MeT's design that relies on three main components: *Monitor*, *Decision Maker* and *Actuator*. The *Monitor* and *Actuator* components can interface with a NoSQL database directly (through the NoSQL interface) and with an *IaaS* (through the IaaS interface). The *Decision Maker* interacts with the *Monitor* and *Actuator* components.

Giving a brief overview, the *Monitor* component gathers important statistics of the running cluster and periodically passes them to the *Decision Maker* component. This component can be considered the core of MeT. Basically, it tries to converge to the same results as the $Manual - Heterogenous$ strategy (see Section 3.1) in an automated way. In that regard it follows a set of stages. The first step involves deciding wether the load cluster is acceptable based on

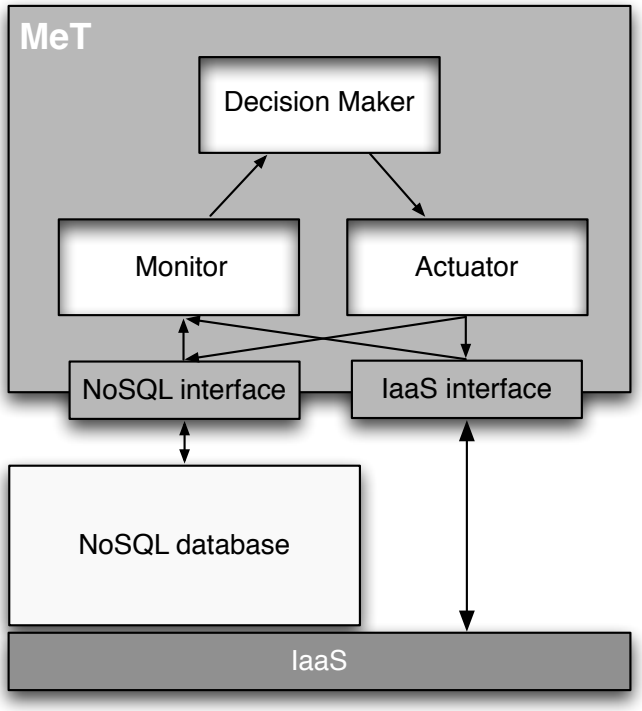


Figure 3.2: MeT's architecture.

the metrics delivered by the *Monitor*. If the cluster is overloaded or underloaded, then it is decided how many nodes must be added or removed from the cluster, respectively. Closely matching the process described in $Manual - Heterogenous$ strategy, this component then classifies each data partition by type of access, clusters them into groups, and proportionally determines the number of nodes to attribute each group. After that, for each group a *greedy algorithm* tries to evenly balance the load in each node. Finally, the *Decision Maker* tries to convey the best way to bring the previously configured cluster to the new computed configuration. This final output is then passed to the *Actuator* that actually implements it in the running cluster. In the following subsections we will describe in detail each component and the algorithms used.

3.2.1 Monitor

The *Monitor* component gathers information about the current state of the cluster (Figure 3.3). Periodically, it collects and maintains data over several cluster

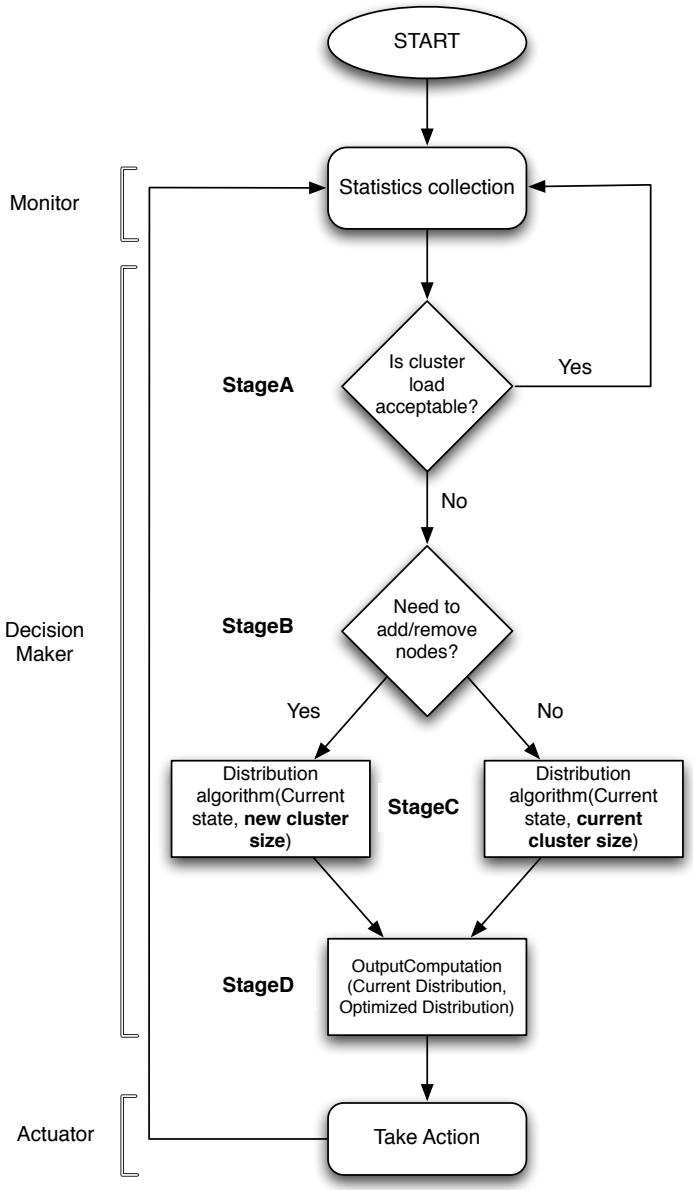


Figure 3.3: MeT's flow chart with particular emphasis on the *Decision Maker* component.

metrics at two different levels: system metrics and metrics specific to the NoSQL database. System metrics are CPU utilization, I/O wait and memory usage. With regard to NoSQL specific metrics, this component needs to keep track of several metrics per node and per data partition. The metrics collected from the NoSQL database must be enough to know the access patterns of the workload.

MeT uses the total number of read, write and scan requests as well as each node locality index. In this regard, the locality index measures the percentage of data that is locally accessible at each node. In other words, it measures the amount of data owned by the node that is locally stored thus not requiring to be fetched through the network when queried.

In order to account for temporary load spikes that could result in poor decisions, we used exponential smoothing [Brown 1963] coupled with storing only the observations after each *Actuator's* action. For each monitoring interval, the last observation is the most important, exponentially decreasing in importance till the first observation. Periodically, all retrieved metrics are delivered to the *Decision Maker* component.

3.2.2 Decision Maker

The *Decision Maker* component is responsible for deciding what actions to take when the cluster is considered to be in a sub-optimal state. As depicted in Figure 3.3 it works following four different stages.

Determine the current state of the cluster (StageA)

StageA (Figure 3.3) begins with the periodical delivery by the *Monitor* component of gathered statistics about the current state of the cluster. Based on those statistics, the *Decision Maker* has to decide whether the load of each node in the cluster is acceptable or not. By acceptable, we mean that the system metrics provided are within certain defined thresholds. Example values for these thresholds are evaluated in subsequent sections.

If the cluster is healthy, the *Decision Maker* remains in StageA (Yes branch of StageA). Otherwise, three data structures are populated to be used in StageB that is immediately initiated. Such data structures are: i) $firstTime$ variable that states whether it is the first time StageB is going to run or not; ii) $subOptimalNodes$ variable which represents the percentage of nodes in a sub-optimal state; iii) *remove* variable that states whether the cluster is under or overloaded.

Decision algorithm for adding and removing nodes (StageB)

In StageB (Figure 3.3), the main task is deciding if it is necessary to add or remove database nodes from the cluster, and if so how many of them following Algorithm 1.

A particular case arises if it is the first time StageB is running ($firstTime$ input). If this is the case, MeT distributes data partitions and heterogeneously configures the current cluster from scratch in what we call an *InitialReconfiguration*. This only happens once.

In subsequent iterations, if the cluster is still in a sub-optimal state, we decide to add or remove nodes. Because we are unable to determine a priori how many nodes we need, those nodes are iteratively added in a quadratic fashion and removed linearly. A quadratic strategy enables a fast response to demand increase by allowing to reach a sufficient number of nodes on a logarithmic number of iterations. That is, the algorithm starts by suggesting the addition of 1 node, and in the following iterations, 2, 4, 8 nodes and so forth, until the load in the cluster is acceptable. Conversely, it removes only 1 node in each iteration, also until the load in the cluster is acceptable. Of course, this strategy may incur a higher provision of temporary resources than necessary. For example, supposing that there is the need for the addition of 8 new nodes. We would only need 4 iterations to reach a sufficient number nodes, but we would have added 15 nodes in the process. In the meantime, if there was not another increase in demand, we would need 7 more iterations to linearly remove nodes until the desired 8 nodes with a total of 11 iterations. On the contrary, if we were adding nodes linearly we would be needing 8 iterations to achieve the desired cluster size. This means that it would take twice as long to reach a point where the number of nodes would be enough to handle the load (from 4 iterations to 8). On the other hand, this also means that we need 3 more iterations to shrink the cluster to the needed size. By using this quadratic strategy we privilege availability and a fast response to sudden load increase.

It should be noted however, that from our experience in the case it is the first time the algorithm is invoked, but the number of sub-optimal nodes is already more than $SubOptimalNodesThreshold$ we proceed straightaway to the addition of nodes. This threshold is a MeT parameter and should be configured according to each system characteristics.

Algorithm 1: Decision Algorithm to add or remove nodes

```
   Data: nodesToChange ← 1
   Input: subOptimalNodes, firstTime, remove
   Result: result
   /* number of nodes to be added or removed from the cluster          */
 1 begin
 2     if subOptimalNodes > SubOptimalNodesThreshold then
 3         result ← nodesToChange
 4         nodesToChange ← nodesToChange * 2
 5     else
 6         if firstTime then
 7             result ← 0
               // InitialReconfiguration
 8         else
 9             if remove then
10                 result ← −1
11                 nodesToChange ← 1
12             else
13                 result ← nodesToChange
14                 nodesToChange ← nodesToChange * 2
15     return result
16 end
```

Finally, the *Decision Algorithm* computes the number of nodes to be added or removed from the cluster, and passes it to the *Distribution Algorithm* in the form of a target cluster size. If there are nodes to be added or removed a new cluster size is computed and passed as a parameter to the next stage. If not, the current size of the cluster is passed as a parameter.

Distribution algorithm (StageC)

The *Distribution Algorithm* corresponds to StageC of the *Decision Maker*'s component (Figure 3.3) and is in fact divided in three parts: *classification*; *node grouping*; and *assignment*. Note that this stage is only reached if the cluster is in sub-optimal state. Even if StageB's result states that there is no need to add or remove nodes, the fact that StageC is running means that a cluster reconfiguration should be attempted in order to improve cluster health.

Classification: data partitions are divided into groups according to access patterns. As stated earlier, we defined 4 groups: read, write, read/write and

scan (see Section 3.1.3). Using the metrics related to the number of write, read and scan requests of each data partition, the *Classification* algorithm assigns each data partition to one of the 4 groups. The assignment of partitions to groups is parameterized with threshold values. In MeT, such values have been obtained by experimental observation and are presented in Section 3.3. In order to accommodate workload changes, metric values obtained for each data partition are refreshed at the beginning of every monitoring interval.

Grouping: the number of nodes to attribute to each group is computed. Each group will be assigned a number of nodes equal to the division of the number of partitions in that group by the total number of partitions, and then multiplied by the total number of nodes available. More formally:

$$\forall g \in G : \frac{\#partitions\ in\ g}{total\#partitions} * total\#nodes$$

Assigment: from node grouping and data partition classification an assigment of data partitions to nodes is established. The assignment is done attempting to balance the load and the number of data partitions in each node. This task falls in a classical problem called *makespan minimization* or *multiprocessor scheduling*, which in turn is related to bin-packing problems. These class of problems are known to be NP-hard [Lenstra et al. 1977] but there are greedy algorithms that provide good results in polynomial time. As a result, we used the greedy algorithm proposed in [Graham 1969], and because we know in advance all data partitions we could use the variant of this algorithm that provides better results - *Longest Processing Time (LPT)*. In short, the *makespan minimization* problem can be defined as: there is a set of parallel processors and a set of jobs with a determined cost; in the LPT version, the algorithm first sorts the set of jobs by decreasing cost and then assigns the largest job to the least loaded processor, until there are no jobs left to assign. In our case, jobs can be translated to data partitions, parallel processors to nodes and the cost of each job to the number of requests of each data partition.

Furthermore, we added a new constraint to the problem to also attempt to balance the number of data partitions assigned to each node. In this regard, the algorithm takes into account node capacity and establishes a maximum number

Algorithm 2: Assignment Algorithm

```
   Data: result ← []
   Input: nodeGroup, dataPartitions, max
   /* max stores maximum number of partitions per node.  Calculated in order
      to balance load.                                                    */
   Result: result
 1 begin
       Data: dataPartitions.sort()
       /* Sort by number of requests in decreasing order.                 */
 2     while dataPartitions.size() > 0 do
 3         partition ← dataPartitions.first()
 4         node ← nodeGroup.getMostEmptyNode()
 5         if node.numberOfPartitions < max then
 6             node.assign(partition)
 7             dataPartitions.remove(partition)
 8         else
 9             result.add(node)
10             nodeGroup.markAsFull(node)
               /* Node already full.                                      */
11     return result
12 end
```

of data partitions per node. This maximum value is estimated by dividing the number of data partitions in the group by the number of nodes in the group.

The assignment algorithm is depicted in Algorithm 2. It should be noted that it has to be called for each group of data partitions.

Output Computation (StageD)

Finally, StageD has the responsibility of determining the *best* way to achieve the targeted cluster configuration. By *best* we mean the one that minimizes node reconfiguration and data partition moves. As depicted in Algorithm 3, it receives as input the current cluster distribution and the distribution suggested by the *Assignment Algorithm*. The first time this algorithm runs, it has no information about the current configuration. At the beginning, we consider that the cluster is homogeneously configured. Thus, the distribution suggestion is passed on to the *Actuator*. This results in an initially heavier full cluster reconfiguration (*InitialReconfiguration*).

In subsequent runs, the algorithm looks at the current distribution of data partitions per node and tries to match it with the new distribution. The process of

Algorithm 3: Output Computation

```
   Data: result ← []
   Input: currentState, optimalState, firstTime
   /* Lists of nodes and correspondent sets of data partitions.          */
   Result: result
 1 begin
 2     if firstTime then
 3         result ← optimalState
 4     else
 5         foreach node ∈ currentState.nodes() do
 6             type ← node.type()
 7             set ← node.partitionSet()
 8             opset ← optimalState.mostSimilar(set, type)
 9             optimalState.remove(opset)
10             result.add((node, opset, type))
11         if optimalState ≠ ∅ then
12             foreach node ∈ currentState.node() do
13                 type ← node.type()
14                 opset ← optimalState.popPartitionSet()
15                 result.add((node, opset, type))
16     return result
17 end
```

matching distributions is made resorting to a set intersection algorithm between sets of partitions. In MeT, the set intersection algorithm is a best effort one. For each set of partitions from the suggested configuration, it tries to find the node that currently holds the most similar set of partitions. The result is an assignment of nodes to configurations and sets of partitions to hold.

If there are new nodes added to the cluster, a set of partitions and a configuration type is assigned to these nodes. The same way, if the targeted configuration does not fully match the current cluster configuration, new sets of partitions and configuration types are assigned to existing nodes. The output of this algorithm is a cluster distribution that minimizes data partitions' reassignment and nodes' reconfiguration.

3.2.3 Actuator

The *Actuator* component carries out the necessary tasks to implement the distribution given by the *Decision Maker*. It is responsible for the actual addition and removal of database nodes. On the one hand, if we are using a *IaaS* system

it means first starting a virtual machine, and only after the NoSQL database. On the other hand, if we are using the NoSQL database directly it has only to start or shutdown the respective processes. The *Actuator* is also responsible for the individual reconfiguration of nodes according to one of the possible four groups defined above. In addition, it assigns the data partitions to those nodes as determined by the *Decision Maker*.

3.3 Implementation

MeT is available as an open source project.[1] In the current prototype we used HBase as the NoSQL database and OpenStack as the *IaaS* platform. OpenStack has gained wide support both from the community and enterprises, and is maturing very quickly [OpenStack Foundation].

At the implementation level MeT has two main parts. It is composed by a Java module and a Python module. The pivotal module is written in Python and comprises the core of the *Decision Maker*, *Monitor* and *Actuator* components of MeT. The Java module is used to gather HBase statistics through the *HBase Administrator* interface within the *Monitor* module of MeT.

Monitoring: The *Monitor* component gathers data about CPU usage, memory usage and I/O wait of the various nodes through Ganglia [Massie et al. 2003]. Regarding the metrics specific of HBase, we collect them through JMX from each *RegionServer*, namely: the total number of read, write and scan requests; the number of requests per second; and an index that measures the data locality of the blocks in the co-located *DataNode*. It also retrieves some metrics of each data partition like the number of read, write and scan requests. The number of scan requests is not available in HBase thus we modified it to calculate and export this metric. All this data is retrieved by MeT's Java module, which interfaces with the Python module through Py4J [Dagenais]. The monitoring intervals are configurable. It is possible to define Ganglia requests periodicity and data history size. Similar to *Decision Maker* parameters, these are also defined in a properties file.

[1] `https://github.com/fmaia/MeT`

Decision Maker parameters: In order for the *Decision Maker* to work, some parameters must be set. Firstly, the classification task of Section 3.2.2 requires a set of threshold values to define types of partitions. Four groups were defined. Data partitions are classified according to the following criteria: i) read, if more than 60% of total requests are read requests; ii) write, if more than 60% of total requests are write requests; iii) scan, if more than 60% of read requests are scan requests; iv) and read-write in every other case. Secondly, $SubOptimalNodesThreshold$ must be configured. In our experiments this threshold was set to 50% of the cluster. This means that if half of the cluster is under heavy load MeT will proceed straightway to the addition of a new node. From our experience, this parameter should be set to 50%, because when most of the nodes in the cluster are under heavy load there is no benefit in the reconfiguration of the cluster without adding new nodes. Moreover, if the cluster in question is subjected to very sudden peak loads it should be adjusted to less than 50% for a faster response to increased demand.

Although we do not envisage that *classification* parameter values can take different values, this may not be the case for other parameters. Consequently, each one of these parameters is configurable in a properties file.

Taking actions: Addition and removal of virtual machines from the HBase cluster is done through the OpenStack interface by the *Actuator*. With regard to node reconfiguration, HBase does not currently provide a mechanism to allow online reconfiguration of a *RegionServer*. That means that every reconfiguration of a *RegionServer* implies its restart. As a result, a full reconfiguration of the cluster is a very costly operation. Bringing the whole cluster down for a full reconfiguration would reduce the amount of time needed for the full reconfiguration, but it would also mean that during that period, all data would be unavailable. Therefore, we use a strategy to incrementally reconfigure the *RegionServers* while maintaining data availability, although with a lower overall throughput. This strategy redistributes the *Regions* from the *RegionServer* that is going to be reconfigured across the remaining nodes that have not been reconfigured yet. Then, when there are no *Regions* left in the *RegionServer*, it is restarted with the new configuration. Finally, the *Regions* determined by *Decision Maker* are assigned to it. If data locality is below 70% for *RegionServers* configured for a write workload and 90%

for all the others, it invokes the *major_compact* operation in order to reestablish data locality. The difference between the two values is that data locality is of more relevance to a read intensive workload and a *major_compact* operation is a costly one. Relaxing the condition for write intensive workloads has the objective of minimizing the load these operations impose on the system. This process is repeated for all *RegionServer*'s reconfigurations.

The concrete values used in our evaluation have been chosen based on experimental observation and our own experience.

3.4 Evaluation

This section evaluates MeT from three perspectives. First we assess if MeT is able to autonomously converge to a performance level comparable to that achieved by a *Manual-Homogeneous* configuration of an HBase cluster. In this first step an YCSB workload is used. Secondly, we evaluate MeT's versatility by exposing MeT to a PyTPCC workload without any kind of customization. Finally, we study MeT's elastic properties in a Cloud environment.

Configuration

In the experiments below, every 30 seconds the *Monitor* component gathers the metrics and sends them to the *Decision Maker* every 3 minutes. The period of 30 seconds is the same used by other approaches [Konstantinou et al. 2012], but the *Decision Maker* is only invoked after having 6 samples to minimize the impact of sudden spikes and take advantage of the exponential smoothing algorithm. The HBase configuration parameters for each group (*Distribution Algorithm* of Section 5.3) are described in Table 3.1.

Node profile	Cache size	Memstore size	Block size
Read	55%	10%	32KB
Write	10%	55%	64KB
Read/Write	45%	20%	32KB
Scan	55%	10%	128KB

Table 3.1: Node configuration profiles.

Convergence

We started by accessing if MeT could autonomously achieve similar performance to a manual heterogeneous cluster configuration ($Manual-Heterogenous$ strategy). The experimental setting is the same of Section 3.1. We used the 6 YCSB workloads described in such section. Then, we configured a HBase cluster with optimized configuration parameters, homogeneous nodes and using the out-of-the-box randomized data placement component ($Random-Homogeneous$ strategy from Section 3.1).

After 2 minutes of ramp-up time, we start MeT. The experiment then runs for 30 minutes logging the throughput from the perspective of the YCSB's clients.

We then compared the results with runs without MeT for the HBase cluster configured with strategies $Manual-Homogeneous$ and $Manual-Heterogeneous$. We picked the run with the best throughput from both strategies from the results presented in Section 3.1. These results are depicted in Figure 3.4.

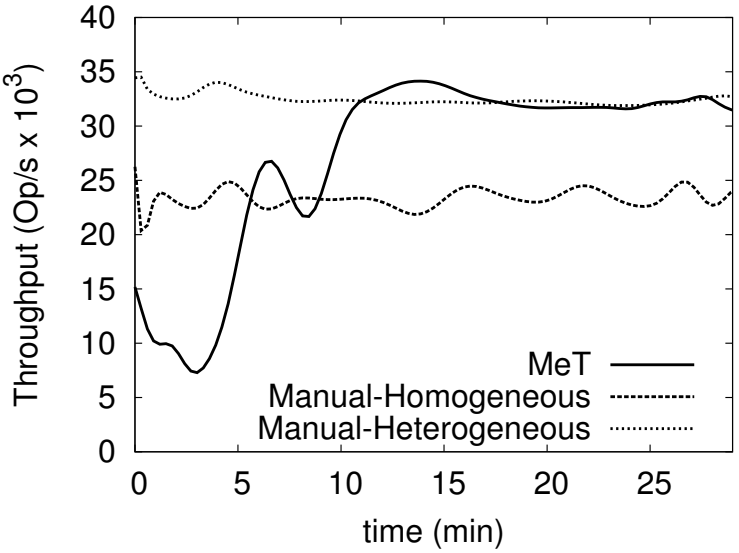


Figure 3.4: Evaluation results

This experiment shows that MeT behaves as expected. It is capable of reconfiguring the HBase cluster *on-the-fly* and achieve similar performance to that of a manually configured cluster. Note that for the same cluster and workload, MeT achieves a significant performance increase: it fully reconfigures a HBase cluster (initially configured with the $Random-Homogeneous$ strategy) in order

to achieve a distribution of data partitions and node's configuration identical to the $Manual - Heterogeneous$ strategy.

The cost of reconfiguration is observable between the 2nd and 8th minute of the experiment (6 minutes). From this overall time, the time taken by target cluster reconfiguration calculations and data mapping is negligible. Restarting the *RegionServers* along with *major_compacts* are the time consuming operations. On the one hand, in our setting a *major_compact* takes roughly 1 minute/GB. On the other hand, most of the impact of reconfiguration on the observed throughput is due to need to restart *RegionServers*, because currently HBase does not allow online reconfigurations. Such feature would allow to greatly decrease this impact. However, by incrementally reconfiguring each *RegionServer* we not only provide continuous data availability, but we also provide reasonable performance with a minimum throughput of 7,500 operations per second. Then, the throughput quickly rises to 20,000 operations per second by the 5th minute and maintains this level of throughput until the reconfiguration is completed by the 8th minute. From this point, the performance is identical to the $Manual - Heterogeneous$ strategy. Even taking into account the reconfiguration cost, within less than 15 minutes the cumulated average throughput using MeT is greater than the default HBase with the $Manual - Homogeneous$ data placement strategy carefully defined by the administrator. These results allow us to state that MeT is able to autonomously reconfigure a running cluster, converging to a cluster configuration and performance level similar to that of a manually configured one.

Versatility

The goal of this experiment is to assess whether MeT could achieve similar results when using a significantly different workload. Moreover, without any change to MeT or its configuration parameters and without any previous knowledge about the workload itself.

For this purpose, we chose PyTPCC[2] an optimized implementation for HBase of the standard OLTP benchmark TPC-C. Note that, while TPC-C standard transactions are expected to have full ACID semantics this implementation offers the isolation semantics provided by HBase: record level atomicity.

TPC-C benchmark attempts to reproduce the behavior of any business in

[2]https://github.com/apavlo/py-tpcc/wiki/HBase-Driver

which sales' districts are geographically distributed along with the corresponding warehouses. There are a total of 9 tables and 5 different types of transactions, and the results are measured in transactions per minute (tpmCs). The default traffic is a mixture of 8% read-only and 92% update transactions and thus is a write intensive benchmark.

The TPC-C database was populated with 30 warehouses resulting in a database of 15GB. TPC-C tables were horizontally partitioned following the usual setting for running TPC-C in distributed databases [Stonebraker et al. 2007]. In that sense, in our experimental setting there were 5 warehouses per *RegionServer*. Each *RegionServer* handles a total of 50 clients.

We ran this experiment on a HBase cluster of 6 *RegionServers*, each configured with a heap of 3 GB, and co-located with 6 *DataNodes*. Similarly to the previous experiment, we used another machine as the master of both HBase and HDFS as well as the Zookeeper instance. PyTPCC's clients were deployed in three other machines amounting to 300 clients (100 client threads per machine), and configured to run for 45 minutes.

This experiment involved three settings: i) a run with a $Manual-Homogeneous$ configuration; ii) MeT starting with a $Manual - Homogeneous$ configuration; iii) and an entire run with the configuration suggested by MeT. The first serves as a baseline and represents the usual way TPC-C runs. It was obtained experimentally, selecting the one that offered the best overall throughput (tpmC), as follows: 50% for the *cache size*; 15% for the *memstore size*; and 32KB of block size. The second setting begins with the same configuration as the first one and after 4 minutes we start MeT to reconfigure the cluster. In the third setting, we used the same distribution and configuration suggested by MeT, but the benchmark was allowed to run for the full 45 minutes without any reconfiguration. Therefore, it represents the maximum throughput that MeT's configuration could possibly achieve.

The results, depicted in Table 3.2, are consistent with those of YCSB i.e. the heterogeneous setting improves the throughput of the $Manual - Homogeneous$ one by 33%. In addition, when comparing the results achieved by MeT and the third setting, the cost of reconfiguration during the experiment is not significant. In fact, around 10 minutes of the total 45 minutes (that is 23%) are due to the phase of ramp-up time (4 minutes) and the initial reconfiguration phase (6

Setting	Throughput (tpmC)
i) $Manual - Homogeneous$	25380
ii) MeT with reconfiguration overhead	31020
iii) MeT w/o reconfiguration overhead	33720

Table 3.2: PyTPCC average throughput results.

minutes). Nonetheless, the overall difference between both settings is just 8%.

The results obtained in this experiment show that MeT is versatile and is able to achieve good results even in the presence of substantially different workloads and without any type of previous knowledge about them.

Elasticity

The experiments conducted so far show that an informed (workload-aware) and heterogeneous configuration of a HBase cluster leads to the best performance. Moreover, MeT is able to autonomously infer and apply such cluster configuration yielding a performance similar to a manually obtained configuration.

In these experiments we go a step further and use MeT as an elastic resource manager that adjusts the size of the cluster according to utilization. To this end, we ran a HBase cluster and MeT on top of an OpenStack deployment. Moreover, we compare MeT's behavior and performance with an existing system called *tiramola* [Konstantinou et al. 2012]. This system, like Amazon's Cloud Watch [CloudWatch] together with Amazon's Auto Scaling [Scaling], automatically provides elasticity to NoSQL databases based on a set of system metrics defined by the client/user of the system. When those metrics reach a threshold, a new node is either launched or retracted from the cluster. Meaning, they are oblivious to the underlying NoSQL system: they just add/remove nodes from the cluster, they do not reconfigure nodes, neither they make data load balancing, nor migrate any data from node to node. We compare MeT with *tiramola* because is the only freely available system.

For this experiment, the HBase cluster is initially configured with seven virtual machines with 3GB of RAM: one for the HBase *Master*, the Hadoop *Namenode* and the Zookeeper in standalone mode; the remaining six for *RegionServers* co-located with *Datanodes*. In every run the initial state is identical: 100% data locality; a replication factor of 2; and data partitions manually balanced on a

homogeneous configuration of the cluster.

In this experiment, we provided each system with a set of YCSB workloads that overloads the initial system. The experiment ran for approximately 60 minutes and was divided in two phases. In the first phase (33 minutes) all clients were active and we observed the throughput and the number of nodes in the cluster.

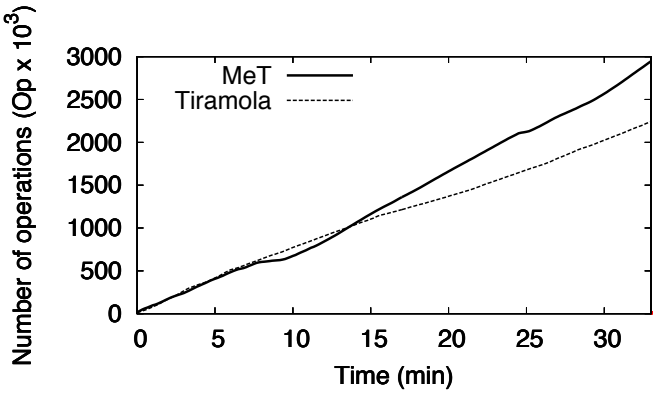
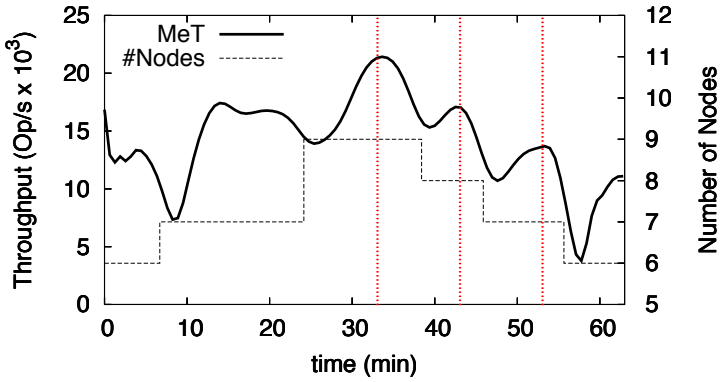


Figure 3.5: Cumulative throughput of MeT and *tiramola* in the first phase of the experiment.

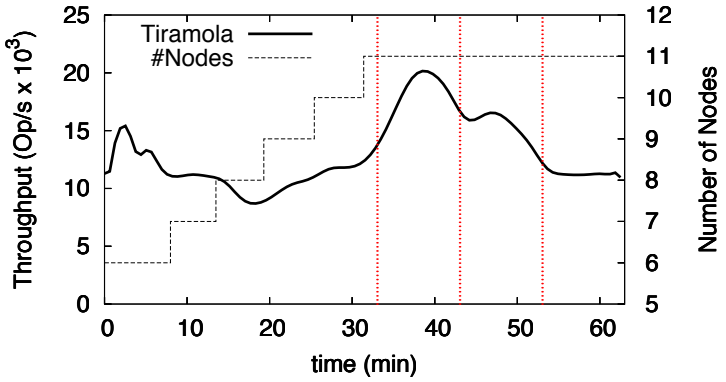
Figure 3.5 shows the cumulative throughput achieved in both scenarios. As it is observable, the HBase cluster managed by MeT outperforms the one managed by *tiramola*. By the end of the first phase MeT has completed more 706,000 operations than *tiramola*, corresponding to a 31% throughput increase. Note that these results are obtained despite the initial MeT reconfiguration cost (from 4th to 11th minute), which starts to pay off after around minute 12. Equally important in a Cloud environment is the amount of resources required to achieve such throughput. This is depicted in Figure 3.6 that shows the throughput evolution (left YY axis) and the number of machines in each cluster (right YY axis).

MeT's throughput is not only superior to *tiramola* but the number of machines is less, requiring 9 machines against 11. Also note that the peak performance achieved by MeT actually corresponds to this scenario maximum achievable throughput of 22,000 operations/second where all YCSB clients are saturated.

Interestingly, even though *tiramola* adds more machines to the cluster there is no significant increase in throughput until the 20th minute. This stems from



(a) MeT



(b) Tiramola

Figure 3.6: Elasticity experiment.

the random balancing and the degradation of data locality, which are precisely addressed by MeT. MeT judiciously balances the cluster and periodically performs a *major compact* for the regions loosing locality and the heterogeneous configuration achieved by MeT increases the cluster throughput by configuring each *RegionServer* accordingly to the workload.

In the second phase of the experiment, we study the systems under resources underutilization. After the 33th minute we progressively switched-off some of the YCSB workloads until there was only one workload active. At minute 33 we turned off $WorkloadE$ and $WorkloadF$, then at minute 43 $WorkloadB$, and finally at minute 53 $WorkloadA$ leaving only $WorkloadC$ running. The experiment results are depicted in Figure 3.6 and workload removal coincides with the vertical lines in the figure.

As can be observed, MeT quickly detects the lower demand and removes

one node from the system. With the progressive lower demand, this process is repeated until the number of nodes is equal to the initial cluster. Please note that, in this experiment we are allowing MeT to release machines each time it detects underutilization, but such behavior is parameterized to avoid, for instance, continuous addition and removal of machines.

On the contrary, *tiramola* only releases resources when every node in the cluster is underutilized. This cannot be parametrized and is due to the homogeneous nature of the *tiramola* managed cluster where removing a single node can divert the load to other already overloaded nodes. The differences in throughput between both systems are due to this behavior, because while MeT is terminating one node and reconfiguring, *tiramola* is just receiving less requests.

3.5 Discussion

In this chapter, we focused on automated elasticity for NoSQL databases. Current approaches to automated elasticity for NoSQL databases look at the different cluster nodes as identical entities. Therefore, elasticity is limited to the decision of adding or removing nodes from the cluster according to demand. Introducing the possibility of having cluster nodes configured heterogeneously proved to allow for better performance and resource usage; an outcome only possible when taking the workload into account.

Following our motivation tests we designed and implemented MeT. The MeT framework provides automated workload-aware elasticity for NoSQL databases. Currently, our prototype is compatible with HBase and OpenStack as the underlying IaaS. Our experiments showed that MeT was able to autonomously reconfigure an HBase cluster without the need to stop it, and achieve similar performance to that of a judiciously and manually configured one. Furthermore, we compared the performance of MeT with an existing system. From this comparison it was possible to see that MeT achieves a cluster configuration that outperforms the cluster obtained using such approach. On top of that, this result was achieved with less resources. In the following chapter, we focus on workload aware data partitioning which is a good complement to MeT system.

Chapter 4

Workload Aware Data Partitioning

In this chapter, we focus on the need for an automated mechanism to find splitting points that take into account the system workload. Current systems (as described in Chapter 2) split data partitions in order to distribute load across cluster nodes. The decision of when to split is made based on a size threshold. However, the splitting point itself is also size based. Typically, data partitions are always split in half. We argue that such splitting impairs load balancing as different data partitions, due to non uniform workloads, may be subject to very different load patterns.

The main problem we address is finding a good splitting point. A good splitting point is the one that splits a set of keys into two new key subsets with similar load. We define a good splitting point in this manner as it leads to better overall load balancing of requests across all data partitions. Therefore, in the following sections we propose and describe a workload aware data partitioning mechanism. The mechanism proposed estimates, in an autonomous way, a splitting point that leads to optimal balance of requests. The algorithm is as simple as effective, and, it is generic and therefore applicable to different NoSQL databases. For clarity of presentation, from now on we will refer to data partitions as *regions* which are no more than sets of keys. We believe it simplifies the description of the mechanism and its implementation is also on HBase.

4.1 Algorithm

With the goal of finding the splitting point we designed an algorithm. An important requisite is that any kind of mechanism we devise does not impose high overhead over the data store system. Doing otherwise would render it highly undesirable. Moreover, it needs to divide the *region* into two *regions* with similar load independently of the request distribution applied to the system. Having these constraints in mind we propose an algorithm that can be run asynchronously and has no impact on the data path. Moreover, as we will see, it achieves highly accurate results with negligible memory and CPU consumption.

Algorithm 4: Split key search algorithm.

```
1  begin
2      foreach Region do
           Data: LowestKey ← ∞
           Data: HighestKey ← 0
           Data: splitKey ← null
3  end
4  On request :
       Data: key ← Request.getKey()
       Data: region ← key.getRegion()
       Data: splitkey ← region.getSplitKey()
5      if key > region.HighestKey then
           Data: region.HighestKey ← key
6      if key < region.LowestKey then
           Data: region.LowestKey ← key
7      if splitKey == null then
           Data: splitKey ← key
8      if key > splitkey then
9          splitkey.increase()
10         if splitkey > region.HighestKey then
11             splitkey = region.HighestKey
12     else
13         splitkey.decrease()
14         if splitkey < region.LowestKey then
15             splitkey = region.LowestKey
16
```

Relying on a simple mechanism to access *region* load it is possible to have a good estimation of the key that splits a *region* into two with similar load. The algorithm, depicted in Algorithm 4, works as follows. The algorithm initiates by estimating that the splitting point is the key of the first request it intercepts for each *region*. By taking into account subsequent requests it will progressively improve its estimation of the splitting point. For each request, if the requested key is smaller than the current estimation the algorithm decreases it. Otherwise, the estimated splitting point is increased. At each request it also updates the smallest ($LowestKey$) and the highest ($HighestKey$) object keys, of which that *region* is responsible for. Such information is useful to know the *region* boundaries.

As the reader easily notices, the increase and decrease methods are not defined in Algorithm 4. This is intentional as their implementation may vary and will impact the performance of the algorithm.

4.1.1 Instantiation

We consider three instantiations of the *increase* and *decrease* methods. The simplest case is to have linear increase and decrease behavior. This means that, for instance, if a request arrives for a key whose value is greater than the current splitting point, the latter will be increased by a constant value. The second instantiation is an exponential function. This means that when two or more steps are done in the same *direction* the step size increases in a quadratic fashion. The final instantiation is achieved by mixing both strategies as described later in this chapter.

Considering these instantiations we set up a few experiments. We considered a key range of 10,000 keys and generated 20,000 key requests that followed a ZipFian distribution. This distribution was chosen as it is representative [Java et al. 2007]. At each request, we looked at the splitting point estimation given by the algorithm. As the distribution was known beforehand, we used the distribution's cumulative distribution function to calculate how the *regions* would be split should such estimation be used. The optimal splitting point corresponds to the point where 50% of the requests fall into each one of the new *regions* i.e. $P(X \leq 50)$ of the cumulative distribution function.

We configured the algorithm with the linear strategy and the results of the experiment are depicted in Figure 4.1.

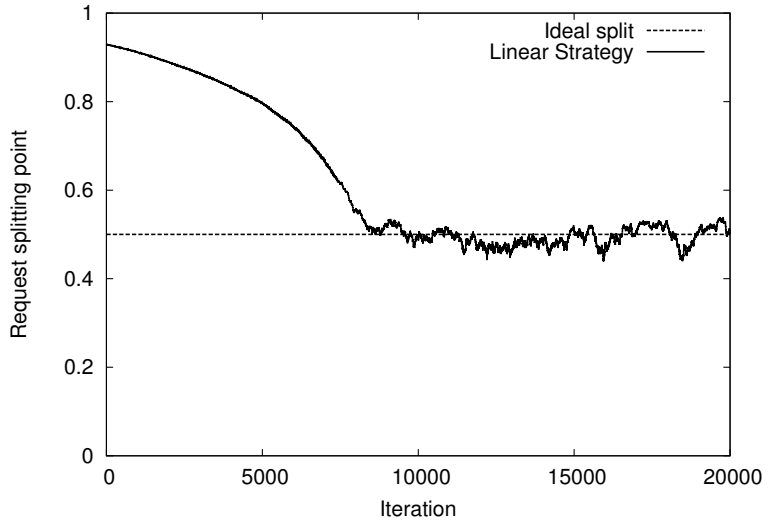


Figure 4.1: Split key search algorithm with linear strategy.

As shown by the experiment results, the algorithm tends to yield good approximations to the ideal split value after 8,000 iterations (that corresponds to 8,000 requests). However, by manipulating the implementation of the increase / decrease methods it is possible to improve the algorithm. The slow convergence of the approach above is due to the fact that, at each request, the algorithm is taking very small steps towards the desired point. Relying on the exponential strategy this can be avoided. As observable in Figure 4.2, the algorithm is now much faster at the expense of stability.

The bottom line is that neither strategy is very attractive. On the one hand, the linear strategy requires a lot of iterations to converge to the optimal split point. On the other hand, the exponential strategy converges rapidly to the optimal split point, but proves to be unstable.

Therefore, our approach is based on the combination of both strategies. The idea is to start with the exponential strategy and, when sufficiently close to the ideal splitting point, change to the linear one. The challenge of this approach is knowing what *sufficiently close* means and how to detect it. To address this problem we try to detect what we call a *PingPong* zone.

Intuitively, if a splitting point is ideal, it means that the probability of a request being for a key smaller than the splitting point is equal (or roughly equal) to the probability of the request being for an higher key. Consequently, the algorithm will fall into a *PingPong* zone where the value of the splitting point

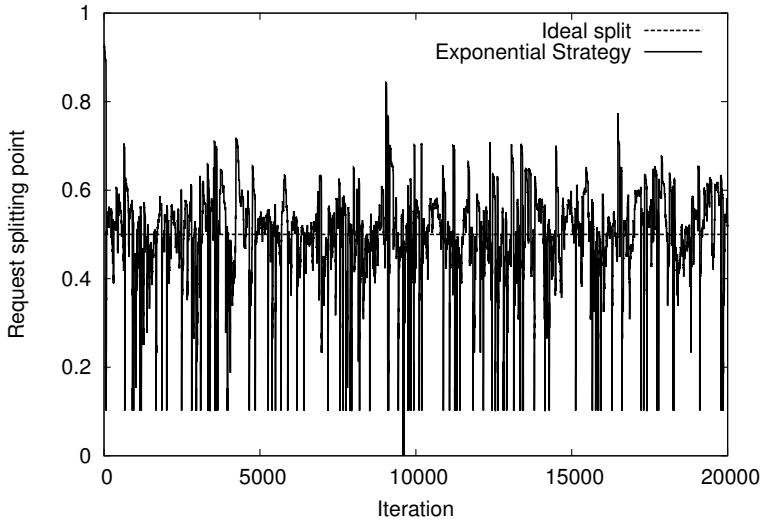


Figure 4.2: Split key search algorithm with exponential strategy.

will be continuously being increased and decreased. When in a *PingPong* zone, the algorithm mutates to the linear strategy.

In order to detect a *PingPong* zone we try to detect consecutive *PingPong* pairs. A *PingPong pair* is a single increase / decrease sequence. The algorithm is configurable in order to define the number of *PingPong* pairs needed to trigger the algorithm mutation. Figure 4.3, depicts the behavior of the algorithm configured with the linear strategy and two mixed strategies. It is possible to observe that a mixed strategy proves to be effective. Moreover, in our experiments we observed that, for this scenario, configuring the algorithm with a small number of *PingPong* pairs allows for very good results.

Another important aspect of the algorithm is that it should achieve good results independently of the request distribution. In Figure 4.4 are depicted the results of an experiment where a Poisson distribution was used. Using the same key range size of the previous experiment, 10,000 unique keys, and 20,000 requests following the Poisson distribution. As it is observable, even for this distribution, the algorithm achieves acceptable results. It is however worth noting that a Poisson distribution is a worst case scenario for finding a good splitting point. It is sufficient for the splitting key to miss the ideal one by a few intervals in order to yield very different splitting ranges.

Finally, we also wanted to evaluate the impact of the key range size in the performance of our algorithm. Figure 4.5 depicts the results of an experiment

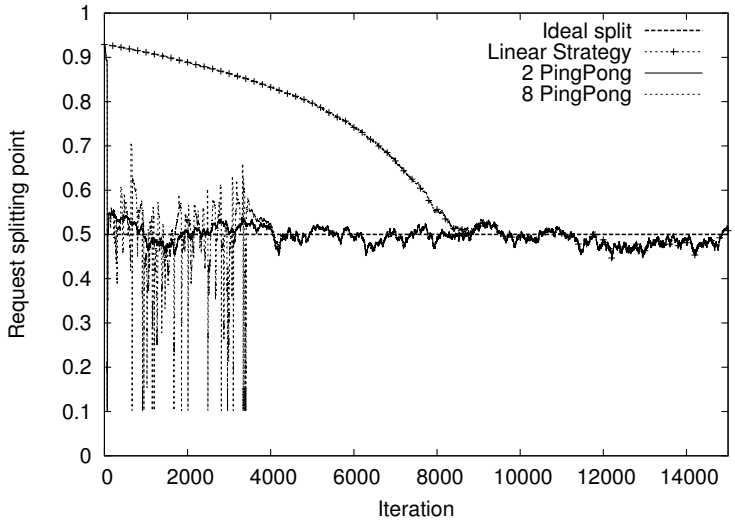


Figure 4.3: Split key search algorithm with PingPong detection.

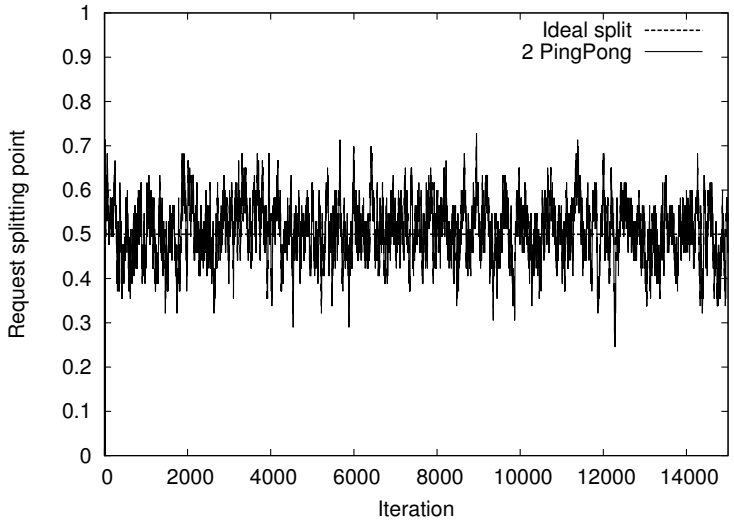


Figure 4.4: Split key search algorithm with PingPong detection for a Poisson request distribution.

where the key range size was increased to 300,000 unique keys. The distribution used was, again, Zipfian. Beginning with some considerations, not depicted in the Figure, the linear strategy is much affected by the key range size, because it depends on the first request to linearly converge to the ideal splitting point. Likewise, the exponential strategy is not much affected by the key range size nor the first request, but once more has some instability. As can be observed, as opposed

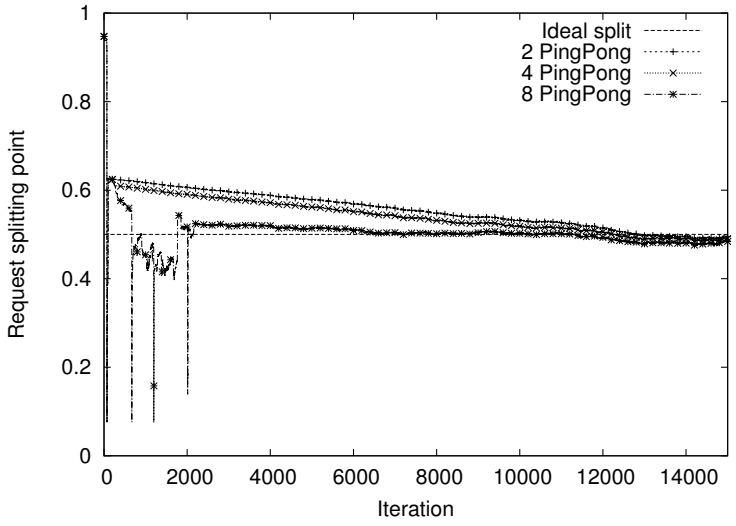


Figure 4.5: Split key search algorithm with PingPong detection for a large key range.

to the experiment with the smaller key range the different configurations of the *PingPong* really impact the behavior of the algorithm. All different configurations quickly get close to the ideal splitting point, but for 2 and 4 *PingPong* pair configuration the switch to the linear strategy occurs too soon, which impacts the convergence of the algorithm. The best results are therefore achieved by 8 *PingPong* pair configuration. Its initial instability is compensated by the closer estimation yield by the exponential strategy and converges to the ideal splitting point in almost the same number of iterations as in the previous experiment. This leads to the observation that for a larger key range size, the number of *PingPong* pairs should be slightly increased.

From the results we can safely conclude that our algorithm provides a good heuristic for finding a suitable *region* splitting point.

4.2 Workload aware data partitioning in HBase

In this Section we describe and evaluate the implementation of our algorithm in HBase. Although the mechanism is generic and applicable to other NoSQL data stores, we will focus on an HBase implementation from now on. In Section 4.2.1 implementation details are described and the evaluation of the mechanism is

presented in Section 4.2.2.

4.2.1 Implementation

Data in HBase is organized into tables which are split into *regions*. HBase splits tables when these reach a certain size. As described earlier, this approach does not lead to good load balancing when data requests obey skewed distributions.

In order to implement our automated workload aware splitting mechanism, we have added the mechanism of the previous Section to the HBase data store itself. Consequently, when HBase is running a splitting point estimation is calculated for each *region* continuously. In particular, it is calculated within each *RegionServer* and exported as a JMX metric accessible through the HBase client interface.

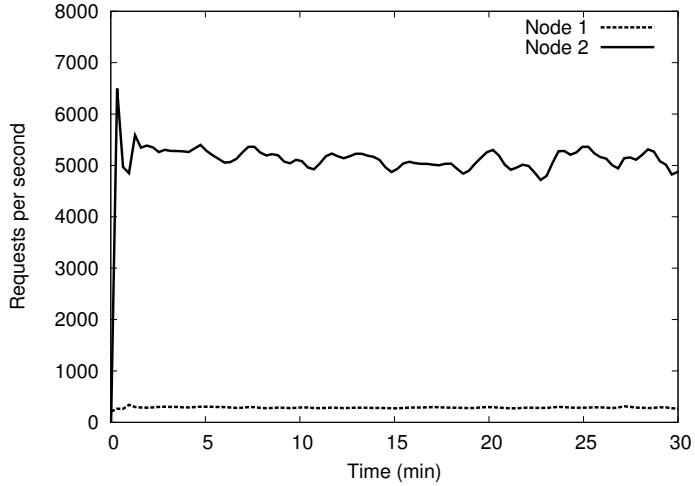
With this implementation it is now possible to split *regions* in a workload aware fashion. It is important to notice that the default load balancer of HBase tries to achieve similar number of *regions* in every node. Using our mechanism, which yields *regions* with similar load, eases such process.

4.2.2 Evaluation

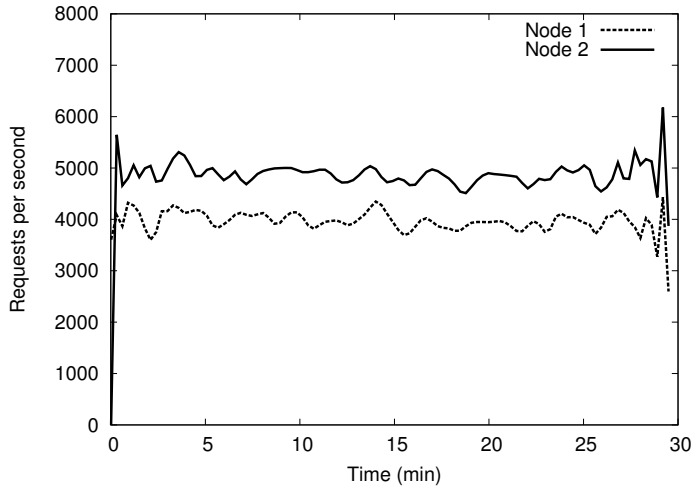
In this Section we present results of our evaluation. The experiment was set up as follows. A HBase cluster was deployed across two nodes. A single table was created and placed on one of the nodes. The table was populated with 1,000,000 records (1 GB of data) using YCSB [Cooper et al. 2010] that was deployed in a separate machine. The same YCSB instance was configured to produce a workload with 80% of read requests and 20% of write requests. Moreover, such workload follows a ZipFian distribution. All nodes used for these experiments have an Intel i3 CPU at 3.1GHz, with 4GB of memory and a local 7200 RPM SATA disk, and are interconnected by a switched Gigabit local area network.

The table was intentionally designed to be too large to be handled by a single node. Consequently, it is split into two *regions* one on each node. At this point, two different scenarios were considered and evaluated. Scenario one corresponds to the out-of-the-box HBase behavior. HBase splits the table into two *regions* of the same size regardless of the access pattern. In scenario two, our splitting mechanism is in place. The initial set up is similar however.

For both scenarios we logged the load imposed on each of the nodes. In order



(a) HBase out-of-the-box with uniform splitting.



(b) HBase with workload-aware splitting.

Figure 4.6: Node load for the two scenarios.

to determine how load was being distributed across both of the cluster nodes. The results are depicted in Figure 4.6.

The split made by the default mechanism not only leads to an highly unbalanced cluster but is also highly infective (Figure 4.6(a)). In fact, node 2 of 4.6(a) is saturated even after the split while node 1 is practically idle. This reduces the cluster capacity to virtually the capacity of the single node. An overloaded node, with this split, remains overloaded as the load moved to the other node is negligible.

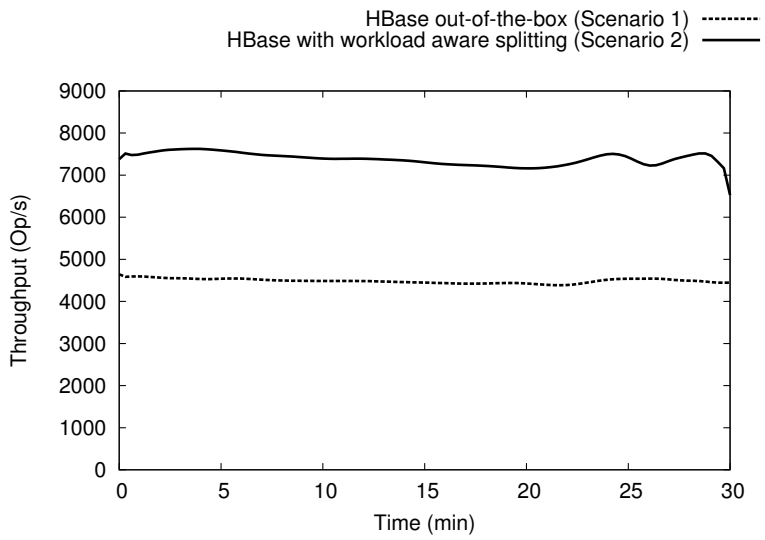


Figure 4.7: Evaluation over HBase. Throughput achieved in scenarios one and two.

In contrast, our approach allows to almost double the overall throughput as depicted by Figure 4.7. Splitting the table using the splitting point given by our algorithm allows for load to be distributed across the two nodes taking advantage of the capacity of both (Figure 4.6(b)).

Note that, after the split the nodes remain under high load. Allowing a second split round and allocating new machines could increase immensely the performance of this cluster. This is not the case when using the traditional split mechanism. Splitting without taking into account the load will always result in an highly unbalanced cluster impairing performance.

The results validate the approach and show that the algorithm proposed is effective in practice. Moreover, it also opens future research paths as automated workload aware data partitioning for NoSQL seems an objective worth pursuing.

4.3 Discussion

Along this chapter we presented a workload aware data partitioning algorithm for NoSQL databases. We evaluated it and proved it to be effective in practice. Although simple, the algorithm proposed is a pragmatic approach to automated splitting point discovery. The algorithm is based on online median estimation and quick convergence is achieved based on a combination of two strategies: linear

and exponential. The algorithm starts with the exponential strategy for rapid convergence and, when sufficiently close to the ideal splitting point, changes to the linear one. This mutation occurs when the algorithm detects what we call a *PingPong* zone.

The results obtained showed the mechanism is effective both for achieving good load balance as well as improving overall performance of HBase. The mechanism presented can greatly enhance MeT system. By identifying the *regions* with highest load, we can then use the mechanism proposed to split the load of that *region* into new *regions*. Then, MeT can rebalance the cluster and achieve an even better cluster configuration.

Chapter 5

Estimating Resource Usage

In the context of cloud computing, predicting and even knowing exactly how a system will behave is very important. Critical decisions on resource allocation, systems configuration or even choice of technology typically require extensive testing, which is still not enough to actually predict the behavior of the system in a real deployment. Moreover, the outcome of these decisions has a direct impact on system cost and effectiveness.

In this chapter we demonstrate that for NoSQL databases it is possible to predict their performance in real-world scenarios even resorting to only a small fraction of the systems resources. Similar to traditional relational databases, NoSQL databases make heavy use of buffer caching, in order to improve the performance of read requests. In order for buffer caching to be effective, the cache hit ratio, which measures the percentage of read requests that result in a cache hit, needs to be as high as possible thus avoiding accesses to slower storage mediums. As a result, the cache hit ratio is directly related to resource consumption. By resource consumption, we mean the amount of main memory used, the number of I/O operations and the amount of memory/disk swapping needed. By predicting resource usage, then it is possible to optimize, prepare, and simulate how systems behave under different conditions.

We take advantage of how the cache hit ratio is directly related to resource usage of NoSQL databases. In clear contrast with relational databases, which due to their inherent complexity require more elaborate models, in this work we show that for NoSQL databases this correlation is actually enough to accurately predict resource usage of any workload. We resort to a resource usage model of read

operations with *uniform* distribution, which for LRU caches represents the case where the cache hit ratio is minimum, and thus the resource usage is maximum. The model built resorting to spline interpolation is hardware dependent, which means it has to be rebuilt when the hardware changes, but it is run only once per hardware configuration and can be used to predict the resource usage for any workload. Using the same approach we show that it is also possible to build a similar model for write operations. Finally, by using both models, we show that it is possible to accurately predict a NoSQL database resource usage, only by knowing two parameters: i) the cache hit ratio and ii) the incoming throughput. From our experiments using either HBase and Cassandra, we accurately predict resource usage for any request distribution and any throughput of read-only and a mix of read and update operations.

5.1 Applications

The work and contributions discussed along this chapter have several interesting applications, namely for predicting optimal software and hardware configurations for NoSQL clusters, and for providing novel load balancing mechanisms for NoSQL databases. In the following, we elaborate further on these possible applications.

Infrastructure configuration prediction: An interesting application is related to the need of predicting what would happen to a NoSQL cluster in different scenarios. For instance, imagining a deployed NoSQL cluster running smoothly, what would happen if all of the sudden a specific data partition had a peak on popularity, and its request rate would increase greatly. Could the running node deal with the load? By using our work a database administrator could easily prepare for such situations. Even by creating automatic rules on the load balancer, so when a data partition reaches a certain request rate, it would automatically assign it to another node. In addition, if one has a running NoSQL cluster and is considering a hardware upgrade, one could use our work to decide what hardware to upgrade to, or even make comparisons between several hardware options. It would require access to at least one machine with the intended hardware specifications. Then, by using our work to build the resource usage model for both

read and write operations, one would know how the system would cope with the expected load, and even how many machines would be needed.

Load balancing: Yet another application of this work concerns load balancing in NoSQL databases. It can be used in conjunction with MeT framework. Instead of trying to minimize the incoming throughput in each node, it is possible to use the contributions described to minimize resource usage in each node. The only additional metric required is the cache hit ratio, which is already exported by NoSQL databases (see Chapter2) or resorting to cache hit ratio estimation algorithms discussed in the literature can also be used [Che et al. 2002]. This combination allows to achieve optimal resource usage. This application is actually implemented and further explained in greater depth in Chapter 6.

5.2 Interdependence of resource usage and cache hit ratio

The cache hit ratio has a great impact on how a system performs and is thus directly related to its resource consumption. By resource consumption we mean the amount of main memory used, the number of I/O operations to distinct storage mediums and the amount of memory/disk swapping needed. The server usage encompasses the CPU time waiting for I/O operations to complete (I/O wait), the time spent on user space (CPU_{user}) and the time spent on kernel space (CPU_{system}).

$$Server_{usage} = I/Owait + CPU_{user} + CPU_{system}$$

With this measure it is possible to have an accurate picture of how the machine is using its resources. Although the I/O wait corresponds to a period when the CPU is free for other computational tasks, we are addressing a specific scenario that focus on a NoSQL database where we cannot achieve a perfect parallelism between I/O wait and CPU usage. In fact, as most operations require network and/or disk resources we must consider I/O wait to accurately represent the cost of such operations in the metric. Thus, if the combined I/O wait and CPU usage reaches 100%, the throughput does not increase by adding more clients.

In order to demonstrate that effectively the cache hit ratio is related to server usage, we set up four different experiments using a HBase deployment and YCSB [Cooper et al. 2010] as the workload generator. These experiments, while not necessarily representative of real-world workloads, cover a wide spectrum of possible behaviors. With these we are able to show a clear and direct relationship between the cache hit ratio and server usage in NoSQL databases.

Experimental setting: In all experiments, one node acts as master for both HBase and HDFS, and it also holds a Zookeeper [Hunt et al. 2010] instance running in standalone mode, which is required by HBase. Our HBase cluster was composed of 1 *RegionServer*, configured with a heap of 4 GB, and 1 *DataNode*. HBase's LRU *block cache* was configured to use 55% of the heap size, which HBase translates into roughly 2.15 GB. It is noteworthy that the *RegionServer* was co-located with the *DataNode*. We used one other node to run the YCSB workload generator. The YCSB client was configured with a *readProportion* of 100% i.e. only issue *get* operations, and with a fixed throughput of 2000 operations per second with 75 client threads so we solely analyze the impact of cache hit ratio in server usage. All experiments were set to run for 30 minutes with 150 seconds of ramp up time and the results are the computed average of 5 individual runs. The server usage was logged every second in the *RegionServer/DataNode* machine using the UNIX *top* command. The *top* command gives us the CPU_{idle} metric that is converted to our $Server_{usage}$ metric in the form:

$$Server_{usage} = 100\% - CPU_{idle}$$

By the end of each experiment, we gathered the global cache hit ratio exported by HBase for the *RegionServer*. All nodes used for these experiments have an Intel i3 CPU at 3.1GHz, with 8GB of main memory and a local 7200 RPM SATA disk, and are interconnected by a switched Gigabit local area network.

First experiment: In this first experiment, a single *region* was populated using the YCSB generator with 4,000,000 records (4.3 GB). This means that the *region* cannot be fitted entirely into the *block cache*: about 1.1 millions records (1.21 GB) remain on secondary memory and must be brought into main memory when requested. There were four different scenarios each with a different configured

request popularity:

1. A *uniform* popularity distribution, that is all records have equal probability of being requested (as mentioned in Section 5.3.1 this is the case where the cache hit ratio is minimum);

2. A *hotspot* popularity distribution, where 50% of the requests access a subset of keys that account for 30% of the key space;

3. A *zipf scrambled* popularity distribution, highly skewed, but because it is scrambled it means the most popular keys are spread across the key space;

4. A *zipf clustered* popularity distribution, highly skewed, and clustered, meaning that the most popular keys are contiguous, which makes them fall in the same cache block.

The results for this experiment are depicted in Table 5.1. As expected, the *uniform* request popularity is the one that achieves the lower cache hit ratio (49%) and thus consumes more server resources (58.35%). On the contrary, the *zipf clustered* request popularity attains the higher cache hit ratio (93%), because the most popular records are clustered and as a result fall in the same block served by HBase's block cache. And because of that it consumes the least Server resources (only 19.28 %) for the same 2000 ops/s.

Distribution	p_{hit}	Average $Server_{usage}$	#Records
Uniform	49%	58.35%	4,000,000
Hotspot	56%	46.19%	4,000,000
Zipf Scrambled	68%	35.91%	4,000,000
Zipf Clustered	93%	19.28%	4,000,000

Table 5.1: Average $Server_{usage}$ and cache hit ratio results under 4 different distributions, for a *region* not fitting in *block cache*.

Second experiment: We set up a second experiment, to demonstrate that the behavior observed in the first experiment was not due to the different request popularities. This experiment is identical to the first one except for the *region* size. This time around the *region* was populated with 2,000,000 records (2.14 GB). This means that the *region* can be fitted entirely into the *block cache*, thus

the expected cache hit ratio is 100%. If there are no differences in server usage between the different request popularities, then the server usage is mainly affected by the cache hit ratio and the incoming throughput. As shown in Table 5.2, the results are as expected. As all data is served only by the *block cache*, the different request popularities become irrelevant to server resource consumption. Therefore, all four distributions use roughly the same amount of server usage.

Distribution	p_{hit}	Average $Server_{usage}$	#Records
Uniform	100%	12.29%	2,000,000
Hotspot	100%	12.14%	2,000,000
Zipf Scrambled	100%	12.92%	2,000,000
Zipf Clustered	100%	12.89%	2,000,000

Table 5.2: Average $Server_{usage}$ and cache hit ratio results under 4 different distributions, for a *region* that fits in *block cache*.

Third experiment: In this third experiment, we show that if the cache hit ratio of two different distributions is the same, then provided that the incoming throughput is also the same, the server usage is identical. From the second experiment this hypothesis is true, however the *region* fitted entirely in the *block cache*. As a result, in this experiment we used the same setting as the first experiment (populated with 4,000,000 records - 4.3GB), but we changed the *hotspot* parameters so its cache hit ratio is the same as the *zipf clustered* distribution, that is 93%. Therefore, we increased the percentage of requests from 50% to 92% that access a subset of keys that account for 30% of the key space. The throughput is again fixed at 2000 operations per second. As seen in Table 5.3, the two distributions consume the same amount of resources despite the differences in the way they access data. As a result, the server usage is independent of the distribution of requests, if the cache hit ratio is the same for a given throughput.

Fourth experiment: Finally, we go a step further and argue that two different distributions, with different data sizes but with the same cache hit ratio, will have the same server resources consumption if subject to the same fixed throughput. Consequently, in this experiment we used the same setting as the first experiment for the *zipf clustered*, again populated with 4,000,000 records (4.3GB), but we changed the number of records of the *uniform* distribution to 2,141,881(2.3 GB)

Distribution	p_{hit}	Average $Server_{usage}$	#Records
Zipf Clustered	93%	19.28%	4,000,000
Hotspot modified	93%	18.93%	4,000,000

Table 5.3: Average $Server_{usage}$ and cache hit ratio results for two different distributions, but with the same cache hit ratio, for a *region* size that cannot fit in *block cache*.

so its cache hit ratio could also be 93%. The throughput is again fixed at 2000 operations per second. From Table 5.4 it is possible to see that the amount of resources used by both distinct distributions is identical. Despite the fact that they have been populated with different data sizes. Therefore, for some throughput all it takes to have an identical server usage is an identical cache hit ratio, regardless of the data size and the distribution.

Distribution	p_{hit}	Average $Server_{usage}$	#Records
Zipf Clustered	93%	19.28%	4,000,000
Uniform	93%	19.76%	2,141,881

Table 5.4: Average $Server_{usage}$ and cache hit ratio results for 2 distributions with different sizes, but with same cache hit ratio.

Correlation between server usage and cache hit ratio: Intuitively, it appears that there is a correlation between the server usage and the cache hit ratio. A correlation test using the *Fisher's z transformation* [Fisher 1921] with the data from the previous experiments, shows that in fact there is a negative correlation for *p-value* < 0.001, making it statistically significant. Accordingly, we can state that for a given throughput the higher the cache hit ratio, the lower the server usage. As a result, we argue that it is possible to estimate the server usage from the incoming throughput and the cache hit ratio. In the following sections we show how this can be done.

5.3 Estimating resource usage of read operations

From the previous section we have shown that the cache hit ratio is related to NoSQL database resource usage: for a given throughput, the higher the cache hit ratio, the lower the server usage. In addition, the cache hit rate reflects not only the data size, but also the underlying distribution of requests which, in combination with an incoming throughput, corresponds to a given server usage. Furthermore, for a fixed throughput this relation is univocal, that is, for some throughput if two distinct workloads consume the same amount of resources, then they must have the same cache hit ratio.

In this section, we show how the server usage of any workload can be estimated simply by knowing its cache hit ratio and incoming throughput, regardless of the distribution of requests and data size. We build on the aforementioned properties to build a tridimensional model, that models the server usage for a NoSQL database, when the cache hit ratio and the throughput vary. It is noteworthy that by collapsing the data size and the request distribution into a single metric, the cache hit ratio, it greatly simplifies the construction of the model. Nonetheless, this model is hardware dependent and has to be rebuilt when the hardware changes or when there are changes in core configuration parameters of the datastore that, for instance, affect the amount of memory allocated to the cache.

5.3.1 Uniform distribution as a building block for server usage modeling

As mentioned in Chapter 2, the LRU replacement algorithm is widely used in numerous systems, particularly in NoSQL databases. One of the properties of LRU caches, is that the uniform distribution represents the case where the cache hit ratio is minimum. Taking into account the negative correlation between the cache hit ratio and the server usage shown in the previous section, it is then possible to deduce that the uniform distribution represents the scenario where the server usage is maximum. Therefore, the uniform distribution represents both a lower bound on cache hit ratio and a higher bound on server usage. As any

other possible distribution will have a greater cache hit ratio, this means that it will consume less resources than the uniform distribution. Moreover, the uniform distribution reduces the training time, since it represents the lowest possible hit rate for a given data size, thereby the time it takes to populate the data in the database is smaller. Therefore, the uniform distribution is a good candidate to be used in building a tridimensional model, which models resource usage of a NoSQL database, when the cache hit ratio and the throughput vary. In order to build such model it would be necessary to test the behavior of the system for every possible combination of cache hit ratio and incoming throughput. Naturally, this is unpractical and actually defeats the purpose of building a prediction model. Instead, we need to test the behavior of the system against a carefuly chosen number of combinations of cache hit ratio and throughput that are representative enough to build the prediction model.

At this point, it is important to note that, for a generic workload generator, it is not possible to define the desired cache hit ratio. Instead we can only set the data size and desired throughput. However, as we have chosen the uniform distribution for building the model we can take advantage of a simple approach proposed by *Che et al.* [Che et al. 2002] that provides an exact estimation of the cache hit ratio of the uniform distribution. This way, we can represent the cache hit ratio by its correspondent data size when building the model. Therefore, in the remainder of this section we will mention data sizes implicitly mentioning their correspondent cache hit ratios.

Let's begin by looking at an illustrative example. Figure 5.1 shows the behavior of incoming throughput when data sizes increase for a fixed server usage percentage[1]. The first obvious observation, also supported by the previous experiments, is that if the data size is smaller than the cache size, then all that matters to server usage is the incoming throughput. In other words, because the cache hit ratio is always 100% in those cases, the only variable affecting the server usage is the throughput. Thus, the throughput is the same for all possible data sizes between 0 and *cache size*. For larger data sizes, the cache hit ratio drops and cache swapping begins, which in turn means that in order for the server usage to stay the same the throughput must decrease. As a result, this is a boundary point (where *data size* equals to the *cache size*). This observation allows to reduce the

[1]Actually it corresponds to the 60% $Server_{usage}$ line in Figure 5.2.

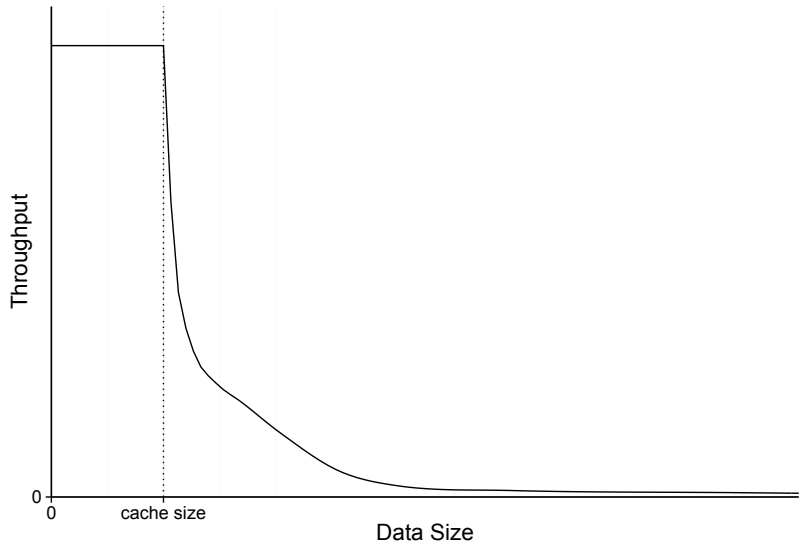


Figure 5.1: Typical relation between cache size and throughput for a fixed $Server_{usage}$.

number of points to calculate for that section of the model as we just need to build the model from that point onwards.

From that point on, we need other heuristics for choosing the points to measure. Naturally, the model will be more accurate as the number of tested combinations increase. However, to keep the number of tested combinations as low as possible, easing the model building process, we need to select them carefully. For instance, for data sizes slightly larger than the boundary point, there is a big drop on throughput in order to resource usage to remain the same. This drop can be more or less abrupt depending on the speed of the secondary memory. Therefore, in order to capture this behavior in the model we need to increase the number of tested combinations of pairs data size and throughput immediately after the boundary point. Conversely, when the data size is largely increased we can be confident of a long and flat tail. Therefore, in that area we do not need to test as many combinations for the model to be accurate.

Model generator

The uniform distribution server usage model is automatically generated resorting to a developed Python script and using YCSB as the workload generator[2].

[2]All the scripts used in this work are openly available at `github.com/fmcruz/suhcr/`

Generally, this script has 2 main parameters: i) a list of cache hit ratios and ii) a list of targeted server usage levels. Hit cache ratios are, as explained earlier, converted to data sizes using the Che's approximation. Then, resorting to a binary search, the script tries to find the necessary throughput of read operations to achieve each specific percentage of server usage for each data size defined as input. Fixing the server usage level and allowing the throughput to be experimentally calculated via the script, allows us to have a representative number of server usage levels without having to test multiple cache hit ratio and throughput combinations in order to have a usable model.

The first point in the construction of the model is usually the boundary point. This point is a pair of data size and throughput. By definition, the data size must be equal to the defined *cache size*, which means the cache hit ratio is 100% for this point. Let's say we are trying to know the boundary point that corresponds to 60% of server usage. What the script does is that tries to find which throughput is needed to achieve the targeted server usage. For every pair of data size and throughput there are 3 runs, each 30 minutes long with 150 seconds of ramp-up time, and the server usage is logged every second in the remote machine where the database is running. Then, it takes the average server usage of the 3 runs and compares it with the targeted server usage. If the average falls within 0.25% of the target server usage (i.e between 59.75% and 60.25%), we found our point and stop. On the other hand, if the server used is higher, then the throughput has to be decreased, and increased otherwise for the following runs. When all the points for a specific server usage level are found, and as mentioned earlier, we resort to interpolation between those points. Specifically, using the monotonic spline interpolation of the R project [3] embedded into the Python script. This process is repeated for each of the targeted server usage levels. This list does not comprehend all of the possible values between 0 and 100%. Instead, by observing through experimentation we noted that a few of them is sufficient (usually 5 equally spaced). Furthermore, by using linear interpolation between the different server usage levels we achieve very accurate results, and ultimately build a tridimensional model that correlates data size, with throughput and expected server usage.

[3] http://www.r-project.org

Model instantiation in our cluster

The server model based on the uniform distribution is the only aspect of this work that is hardware dependent. In fact, whenever the hardware changes the model is rebuilt to reflect the possible differences in hardware behavior. The automatic server model generator was used on our own cluster, each machine with an Intel i3 CPU at 3.1GHz, with 8GB of memory and a local 7200 RPM SATA disk. The generator was configured to use HBase 0.98.3, with one node acting as master for both HBase and HDFS, and it also holds a Zookeeper instance running in standalone mode. There was one *RegionServer* in other machine, configured with a heap of 4 GB, and co-located with a *DataNode*. HBase's LRU block cache was configured to use 55% of the heap size, that HBase translates into roughly 2.15 GB. Other node was used to run the generator.

The generated server model is as depicted in Figure 5.2. There were defined 10 different cache hit ratios: 100%, 95%, 90%, 80%, 70%, 60%, 50%, 40%, 25%, and 15%. These cache hit ratios were then transformed in their data size equivalents to be used as input in the model generator. The first point is the boundary point corresponding to 2,000,000 of YCSB records. As previously stated, for data sizes slightly larger than the *cache size* we need to increase the density of points tested to ensure the model is more accurate. Thus, the following point is only a 5% decrease, and next 6 points are decreases of 10% in the cache hit ratio. On the other hand, predicting a flat long tail from that point on, we just defined 2 points much more apart from each other, 25% and 15% of cache hit ratio, corresponding to 8,000,000 and 12,000,000 records.

As can be seen in Figure 5.2 the solid lines correspond to the 5 targeted levels of server usage, namely 80%, 60%, 40%, 20% and 5%. It is general practice in frameworks for automated elasticity of NoSQL databases [Konstantinou et al. 2012] that the rule governing the addition of new nodes indicate 80% as the maximum usable CPU before a new node is needed in the cluster. This is an empirical higher bound on usable CPU to accommodate operating systems processes, account for possible load spikes and compactions. Therefore, the highest defined level was 80%. However, our model generator can be configured to model any targeted CPU usage.

When eventually the generator has finished searching for the throughput needed to reach the targeted levels of server usage for the various data sizes,

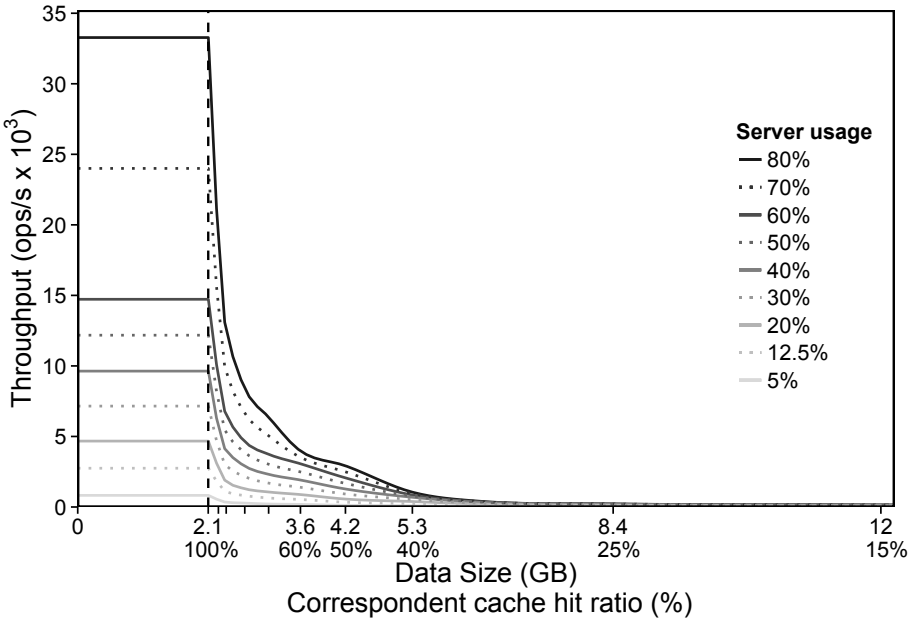


Figure 5.2: Instantiation of the server model for read operations based on a uniform distribution.

it then interpolates the data that resulted in the represented continuous curves. Finally, so the server model can be finished we just need to do a final and linear interpolation between these curves. The curves that correspond to the linear interpolation are represented by dotted lines for the server usage levels of, namely, 70%, 50%, 30% and 12.5%, which are example levels.

Model accuracy

Revisiting the first experiment of Section 5.2, we can now use the generated model to estimate the server usage for the different distributions. The results are depicted in Table 5.5. As can be seen, the estimated server usage is almost the

Distribution	Observed *usage*	Estimated *usage*	Accuracy
Uniform	58.35%	58.35%	100%
Hotspot	46.19%	45.87%	99.31%
Zipf Scrambled	35.91%	36.29%	98.94%
Zipf Clustered	19.28%	19.15%	99.33%

Table 5.5: Observed average $server_{usage}$ and Estimated $server_{usage}$ results under four different distributions.

same as the observed average server usage, despite all four different distributions with very different cache hit rates. It should be noted that, as expected, the approach predicts the server usage of the uniform distribution with accuracy of 100% due to the similarity between the input usage levels of the model and the ones used in the test. Moreover, the generated model proves to accurately estimate the server usage of any distribution with an accuracy no lower than 98%, only by knowing two parameters: i) cache hit ratio; and ii) incoming throughput.

Training

The model construction can be a time consuming task: we observed about 5 days using a single machine. However, to speed up the model's construction, the script can be run simultaneously on different machines provided the hardware is exactly the same throughout the cluster. For the Cassandra model used in Section 5.5 we leveraged a 3 node cluster to reduce model construction time to slightly over 2 days. The reduction was not completely linear because the script uses the following optimization: for consecutive data points the maximum throughput for a server usage level is at most equal to the data point preceding it. In other words, we know that when using the same access distribution if the cache hit ratio decreases then the needed throughput to meet a certain level of server usage must be equal to (if the records fit all in the *block cache*) or lesser than the previous data point. As a result, we use this property to provide a higher bound to the binary search that reduces the search space. However, in the current version, the script launches independent processes and each process has only a local view of the overall model. This would be solved with a distributed version of our script so each process has access to the global view of the model as it is being created.

It should be noted that the construction of the model could be moved to runtime. The model would possibly not be based on the uniform distribution, but instead on distributions observed in the cluster. The only shortcoming is it would likely take a longer time to have a complete model depending on the variety of workloads each node observes and the number of nodes composing the cluster.

Discussion

The approach described in this section allows to accurately estimate the server usage resorting to an offline trained model based on the uniform distribution. Using the cache hit ratio and the incoming throughput as the only parameters that affect resource utilization may appear oversimplifying. Specially, when taking account related approaches to usage prediction in RDBMS. However, key-value datastores are fundamentally different from relational databases. In order to attain high scalability, high throughput and high availability, these datastores offer a simple key-value interface based on put and get operations without providing multi record atomic operations nor complex operations like joins and aggregations. On the other hand, RDBMS must cope with a large number of concurrent and lock-prone ACID transactions and need different and more complex models for resources, such as CPU, RAM, disk I/O and database locks. These differences allow our simple but effective technique to work. In fact, as shown in this Section and in Section 5.5 this technique is valid and achieves high accuracy to estimate the resource usage of any not previously trained workload. The empirical intuition of why other parameters, such as the I/O costs, do not need to be considered separately is because they are already concealed in the training model. Taking a closer look into the behavior of each distribution in the first experiment, and decomposing the overall throughput into operations hitting the cache, and operations that are misses, we have:

- *Uniform* - 49% of cache hit ratio; thus 980 ops/s are cache hits, the remaining 1020 ops/s miss the *block cache*;
- *Hotspot* - 56% of cache hit ratio; thus 1120 ops/s are cache hits, the remaining 880 ops/s miss the *block cache*;
- *Zipf Scrambled* - 68% of cache hit ratio; thus 1360 ops/s are cache hits, the remaining 640 ops/s miss the *block cache*;
- *Zipf Clustered* - 93% of cache hit ratio; thus 1860 ops/s are cache hits, the remaining 140 ops/s miss the *block cache*.

By looking at the average resource usage for each distribution, it is obvious that the cost of a cache miss is greater than the cost of accessing the *block cache*. This implies that the server usage for read operations can be decomposed as the

sum of two costs: $Usage_{read} = Usage_{hit} + Usage_{miss}$ The $Usage_{hit}$ is the cost of only accessing the cache, while the $Usage_{miss}$ represents the cost of a miss in the cache. It covers not only the cost of bringing a block into the cache (either from main memory or disk), but also the cost of discarding the least recently used data to make room for the new data block. Thus, when two workloads have identical cache hit ratios and identical incoming throughputs, it means that both workloads have the same number of operations hitting the cache and the same number of operations missing the cache. As a result, once two workloads exhibit the same $Usage_{hit}$ and $Usage_{miss}$, ultimately exhibit the same server usage. For instance, when the *uniform* distribution was populated with only 2,141,881 records to achieve a cache hit ratio of 93%, it ended up with 1860 ops/s hitting the cache, while the remaining 140 ops/s were cache misses. Making it equivalent, as shown, in terms of server usage to the *Zipf Clustered* distribution. But, the total number of records of the *Zipf Clustered* distribution is almost the double (4,000,000 records). However, due to the skewness of this distribution the 140 ops/s touch a very small fraction of the data, making the likelihood of going to disk similar to the uniform case, and ultimately making the prediction accurate.

5.3.2 Scan operations

In NoSQL databases that follow a hierarchical architecture such as HBase, scan operations allow to read multiple records in batch, which allows to reduce load in terms of I/O and network. The scan operation requires a start and a stop key and then all rows in the given range are read. In fact, in HBase the read of a single row is an instantiation of a scan operation but limited to a single row. As a result, the process used to estimate the resource usage of single row read operations can be extended to estimate the resource usage of scan operations over multiple rows.

As stated earlier, in HBase keys are stored in lexicographic order and thus clustered in blocks. So HBase takes the start key and finds the correspondent block. If the block is not already in the block cache, it reads the block to the cache from secondary storage. Contrasting with a single row read operation, the scan iterates through the keys within the block until the stop key is found. However, if the stop key belongs to another block HBase iterates through the

subsequent blocks until the stop key is found. When estimating the resource usage of scan operations we have to take into account the cost of iterating over the key range ($Cost_{ReadNextRow}$) as well as the cost of reading a block from block cache ($Cost_{BlockFromBlockCache}$) and the cost of reading a block not present in the block cache ($Cost_{BlockOutBlockCache}$). There is yet another cost we must take into account: in a single row read operation the cost of filtering and returning several columns is negligible that is, it does not matter if the single row read operation is requesting just one column from the *ColumnFamily* or requesting all columns. This happens because for a single row, the filtering cost is insignificant when compared to the other costs. However, when iterating over several rows this filtering cost adds up and has to be taken into account ($Cost_{ReadNextColumn}$).

As a result, the process to estimate the resource usage of scan operations is divided into two parts. In the first part we use the exact same procedure described in the previous subsection. In the other words, we build a server usage model based on the uniform distribution with scan operations limited to one row and one column. The instantiation of the model in our cluster is depicted in Figure 5.3.

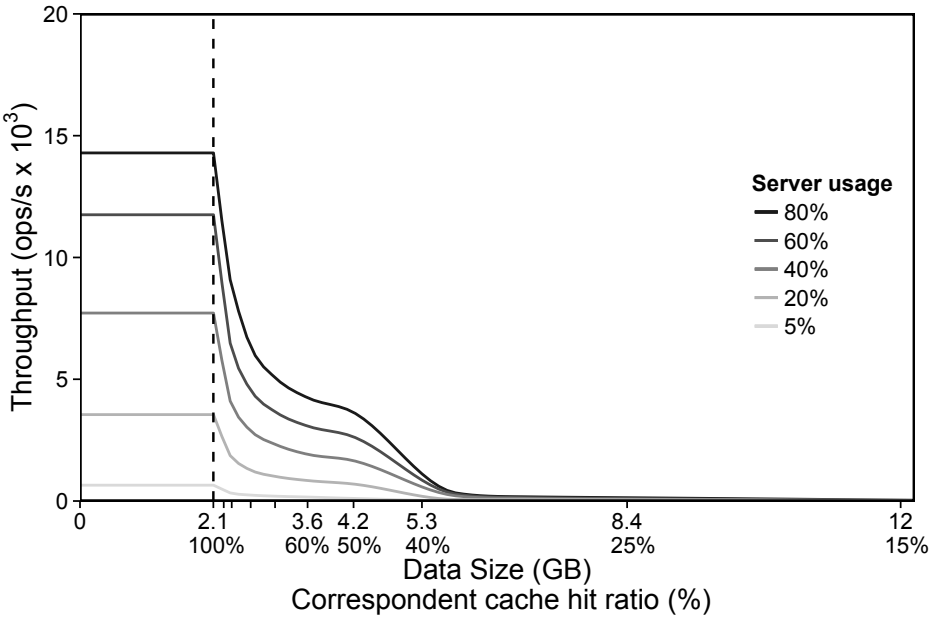


Figure 5.3: Instantiation of the server model for scan operations based on a uniform distribution.

In the second part of the process we can estimate the resource usage of scan operations by adding the aforementioned costs to the model using the following equation:

$$\begin{aligned} ServerUsage_{scans} = ServerUsage_{model} + \#Scans\times \\ (ServerUsage_{ReadNextRow} + ServerUsage_{ReadNextColumn} \\ +ServerUsage_{BlockFromBlockCache} + ServerUsage_{BlockOutBlockCache}) \end{aligned}$$

As can be seen from the equation, first we use the generated model to determine the server usage for the number of observed scan operations. Then, we extend the model by multiplying the number of scan operations by the summation of each of the components that are defined as follows:

$$ServerUsage_{ReadNextRow} = Cost_{ReadNextRow} \times (ScanLength - 1) \quad (5.1)$$

$$\begin{aligned} ServerUsage_{ReadNextColumn} = Cost_{ReadNextColumn} \times (\#Columns - 1) \\ \times ScanLength \end{aligned} \quad (5.2)$$

$$\begin{aligned} ServerUsage_{BlockFromBlockCache} = Cost_{BlockFromBlockCache} \\ \times (BlocksTouched - 1) \times p_{hit} \end{aligned} \quad (5.3)$$

$$\begin{aligned} ServerUsage_{BlockOutBlockCache} = Cost_{BlockOutBlockCache} \\ \times (BlocksTouched - 1) \times (1 - p_{hit}) \end{aligned} \quad (5.4)$$

As stated previously, all equations from 5.1 to 5.4 depend on a set of costs, namely $Cost_{ReadNextRow}$, $Cost_{ReadNextColumn}$, $Cost_{BlockFromBlockCache}$ and $Cost_{BlockOutBlockCache}$. All of these costs are constants and must be obtained experimentally, moreover they are hardware dependent. In addition, the computation of the server usage also depends on the characteristics of the scan operation: i) the scan length, that is how many rows are read; ii) the number of columns to be retrieved; iii) the number of blocks the scan touches, which is also dependent on the scan length; and finally iv) the cache hit ratio. It should be noted that equations 5.3 and 5.4 are complementary, in the sense that they both depend on the cache hit ratio and, for instance, when the cache hit ratio is maximum (100%) then no blocks are read out of the block cache and the $ServerUsage_{BlockOutBlockCache} = 0$. Another important aspect is that if the observed scan operations only return one row and one column then, as expected the

$ServerUsage_{scans} = ServerUsage_{model}$. Because, in this case $ScanLength = 1$, $\#Columns = 1$, and $BlocksTouched = 1$ then all equations from 5.1 to 5.4 equal 0 and thus the $ServerUsage_{scans} = ServerUsage_{model}$.

Some of the necessary metrics regarding the scan operation are not made available by HBase. So we included in HBase metrics engine a new set of scan related metrics for each *region*, namely: distinguish between single row read operations and scan of multiple rows; and the total size of the scan operations i.e how many rows were read by all scan operations. These modifications impose no overhead to HBase.

5.4 Estimating the resource usage of update operations

Although workloads are generally dominated by reads, most applications also have updates. Despite the fact that it is not the main focus of this work, we apply a similar approach to update operations. Updates and writes can be used interchangeable, because in key-value datastores, such as HBase and Cassandra, updates and new writes are append-only, so they follow the same write path.

Updates in these datastores are first written to main memory before being flushed to disk. Therefore, the resource cost of an update is essentially related with the operation of writing the update to main memory and, from time to time flushing it to secondary memory. As a consequence, contrary to read requests, updates are mostly independent of the request distribution and current data size. In addition, because the write path and the read path in a NoSQL database are substantially separated, the overall server usage can be defined as the sum of the usage related with read operations, both single row and multiple row operations, and the usage related with update operations:

$$ServerUsage_{overall} = ServerUsage_{read} + ServerUsage_{update}$$

The server usage related to read operations is estimated using the procedure described in Section 5.3. On the other hand, the server utilization related to update operations is estimated in a different way. As updates are independent of the request distribution and the data size, creating a model to predict the server

usage of update operations is simpler than the read model counterpart. The only variable affecting the server utilization is, thus the write throughput.

5.4.1 Model generator

Analogous to the model generator for read operations, we used a Python developed script to generate the server usage model for update operations. It also uses YCSB as the workload generator, but this time configured for updates. As the update model only depends on the throughput, the script has only one main parameter: a list of targeted update throughput points to test. Evidently, the YCSB must also be loaded with an initial data size, but it is not relevant to the updates server model. For every element of targeted update throughput there are 3 independent runs, each 30 minutes long with 150 seconds of ramp-up time, and the server utilization is logged every second in the remote machine where the datastore node is running. When all the defined points are finished, we also resort to interpolation between those points. Again, we use the monotonic spline interpolation of the R project embedded into the Python script, which achieves very accurate results and allows to diminish the number of data points needed to test.

5.4.2 Model instantiation in our cluster

The server model based on updates is also hardware dependent. Meaning that when the hardware changes the model has to be rebuilt. Like the server model of read operations, the automatic server model generator was used on our own cluster, using the exact same setting. The generated server model for updates is depicted in Figure 5.4. There were defined 28 different targeted update throughputs from 5 updates per second to 10,000 updates per second. For increased accuracy, the first 10 targeted throughputs fall within the interval of 5 to 1000 updates per second. From that point on, there were 500 increments until 10,000 updates per second, which is the point where the server usage reaches 80%. As can be seen the server utilization for update operations grows linearly with the increased throughput. In Section 5.5, it is shown that it is possible to estimate the server utilization when simultaneously dealing with read and update operations.

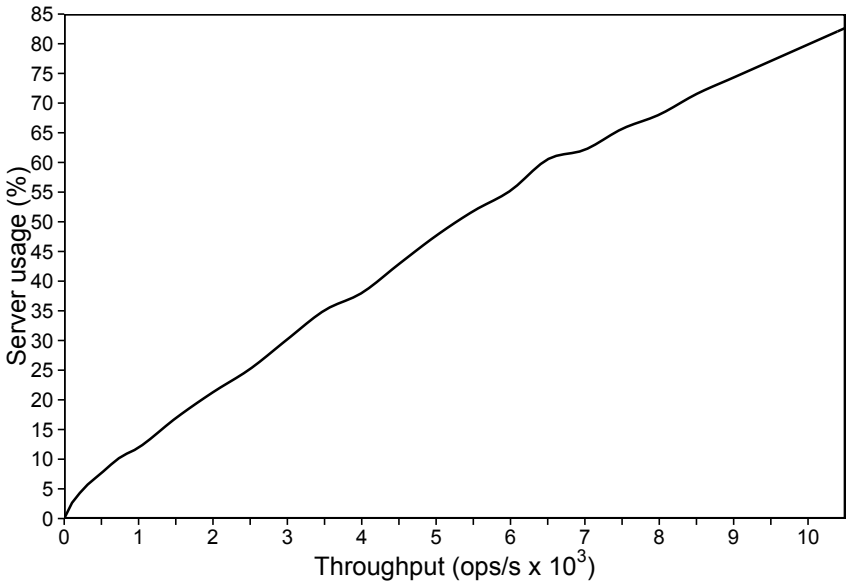
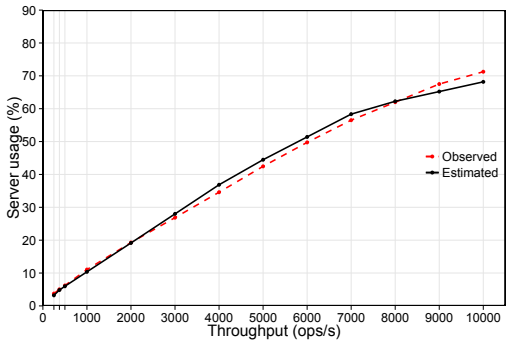


Figure 5.4: Instantiation of the model for update operations.

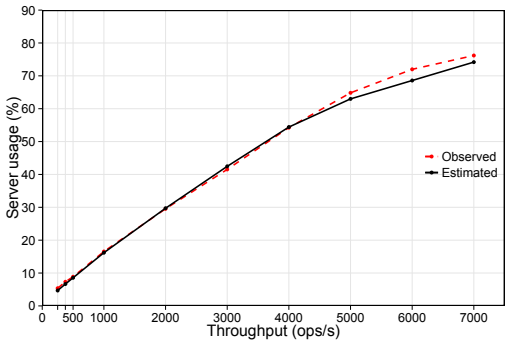
5.5 Validation

This section validates the proposed approach using two different NoSQL databases: HBase and Cassandra. For HBase, we first used a single *region* to show that we can accurately predict the server usage of not previously trained workloads, for a wide spectrum of read throughputs. Secondly, we show that in a multi-tenant setting when two or more *regions* with different request distributions are in the same *RegionServer*, we can still accurately predict the server usage. Moreover, we prove that we can also predict the server usage when there is a mix of read and update operations. Finally, we show that the proposed approach is valid even using different hardware specifications. In this regard, we instantiated the read model as described in Section 5.3 for a different hardware specification. As the process is identical, for simplicity we omit the depiction of the generated models.

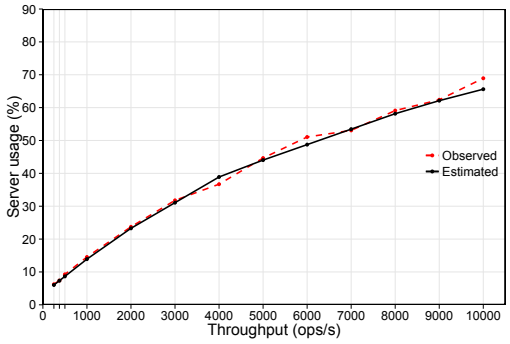
In this section, we also demonstrate that the approach is generic and extensible to other datastores, namely Cassandra. Thus, again using the model generator described in Sections 5.3 and 5.4 we instantiated the read model and the update model for Cassandra.We leveraged a 3 node cluster to reduce model construction time to slightly over 2 days. Once more, for simplicity we omit the depiction of the generated models. Nonetheless, since Cassandra was subject to the same set



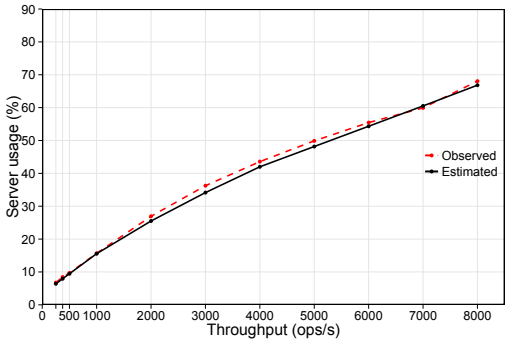
(a) HBase: 4 million records; *zipf clustered*.



(b) HBase: 3 million records; *zipf scrambled*.



(c) Cassandra: 4 million records; *zipf clustered*.



(d) Cassandra: 3 million records; *hotspot*.

Figure 5.5: Experiments for read-only operations in HBase and Cassandra.

of tests as HBase (except evidently for the multiple *region* experiments as there are not *regions* in Cassandra), we use them to estimate the server usage in the different experiments.

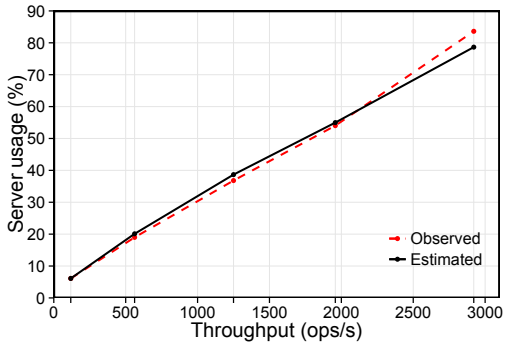
5.5.1 HBase

Configuration: In the experiments below we used the exact same setting as in the experiments of Section 5.2. Briefly, we ran all experiments in our cluster, each machine with an Intel i3 CPU at 3.1GHz, with 8GB of memory and a local 7200 RPM SATA disk. HBase 0.98.3 was deployed, with one node acting as master for both HBase and HDFS, while holding a Zookeeper instance running in standalone mode. There was one *RegionServer* in another machine, configured with a heap of 4 GB, and co-located with a *DataNode*. HBase's LRU block cache was configured to use 55% of the heap size. YCSB was used as the workload

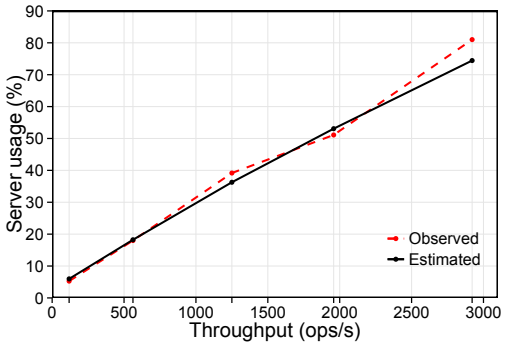
generator, with the client running in other node. All experiments were set to run for 30 minutes with 150 seconds of ramp up time, and the server usage was logged every second in the *RegionServer/DataNode* machine, using UNIX top command. For every data point there were 3 independent runs, and the results presented are the computed average.

Single *region*: We began by assessing whether the proposed mechanisms can accurately predict the server usage, when it comes to other distributions than the uniform distribution. Therefore, we populated the HBase instance using the YCSB's client with 4,000,000 records (4.3 GB). Then, the YCSB's client was configured to use the *zipf clustered* distribution with 100% read operations, and for a fixed throughput ranging from 250 ops/s to 10,000 ops/s. We also wanted to validate what happens when using a data size not used in the model generator. As a result, we populated the HBase cluster with 3,000,000 records (3.15 GB), and this time using the *zipf scrambled* distribution, which yields a much lower cache hit ratio (78.8%). As a result, the configured read throughput ranged from 250 ops/s to 7,000 ops/s. The results for each experiment are depicted in Figure 5.5(a) and in Figure 5.5(b). They show the estimated server usage compared to the observed one. The estimated results are drawn from our approach using the generated model for read operations, and observing the cache hit ratio as provided by HBase exported metrics. As expected, the estimated server usage in both experiments is very similar to the observed counterpart.

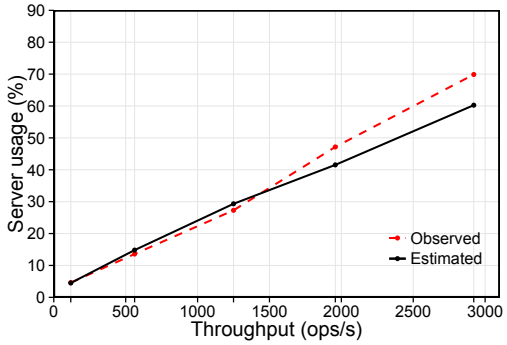
Multi-tenancy: In this step of validation, we assess if we can accurately estimate the server usage in a multi-tenant setting. In HBase, we can simulate this setting with two or more *regions*. As mentioned previously, they compete for the cache, and thus the cache hit ratio is affected by their request distribution and the probability of being accessed. Therefore, we began with 2 independent *regions* in the same *RegionServer*. We configured the read throughput to 2,000 ops/s, and then variated the probability of each *region* being requested from 10% to 90%, yielding 9 possible different combinations. For instance, if one of the *regions* has access probability of 10%, it means that is receiving 200 ops/s while the other one is getting the remaining 1800 ops/s. Both *regions* were populated with 2,000,000 records (totaling 4.3 GB). The YCSB client was configured to issue read requests following a uniform distribution for *Region A*, and *zipf clustered*



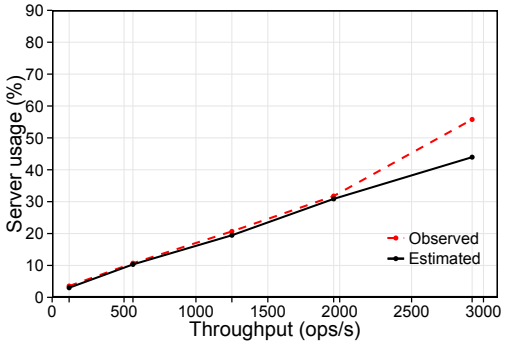
(a) 90% read and 10% update.



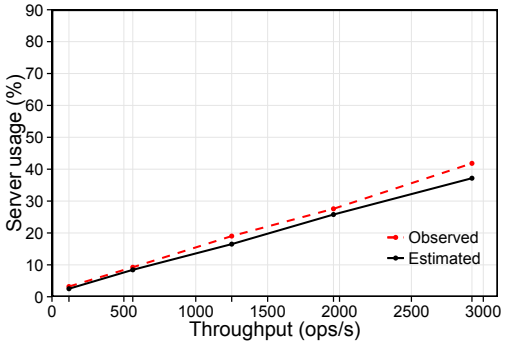
(b) 80% read and 20% update.



(c) 50% read and 50% update.



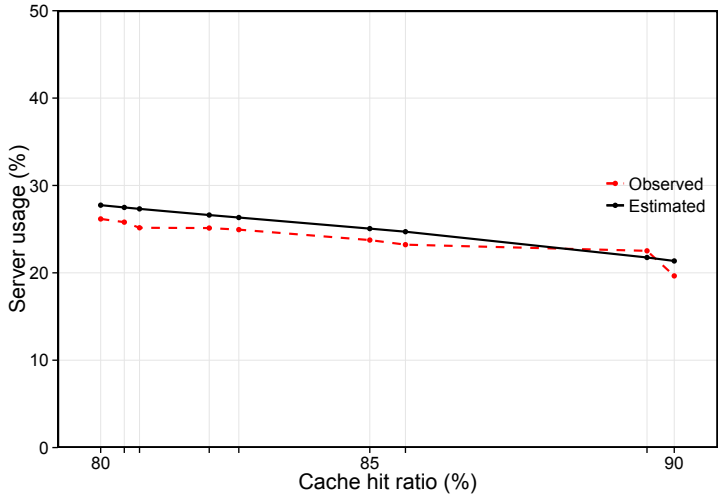
(d) 20% read and 80% update.



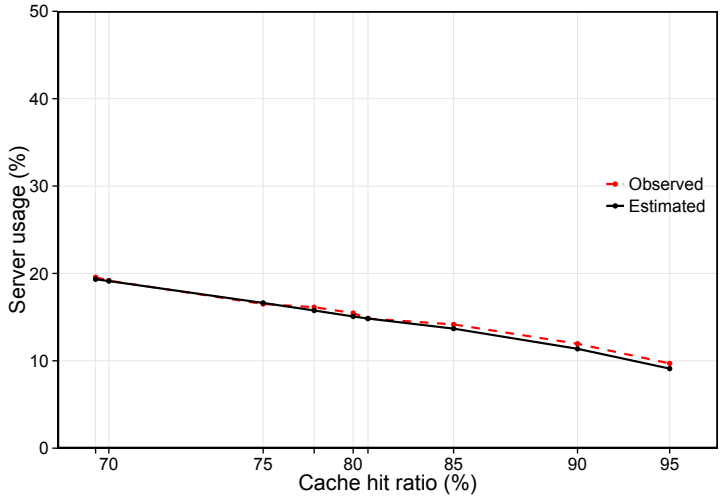
(e) 10% read and 90% update.

Figure 5.6: Read and update operations mix experiments in HBase.

distribution for *Region B*. As depicted in Figure 5.7(a), once more, the estimated server usage accuracy is very similar to the observed one. It is noteworthy that the first point corresponds to the minimum cache hit ratio. The cache hit ratio is at its minimum when both *regions* compete equally for the cache. Thus, they have equal probability of access, that is 50%. On the other hand, the maximum



(a) Experiment with 2 *regions*.



(b) Experiment with 3 *regions*.

Figure 5.7: Multi-tenancy experiments in HBase.

cache hit ratio occurs when the *zipf clustered* distribution is more likely to be accessed - 90%.

The same behavior can be seen when instead of 2 *regions*, there are 3 *regions*. In this experiment *Region A* and *Region B* were populated with 1,000,000 records, and *Region C* with 2,000,000 records (totaling 4.3 GB). YCSB was configured to issue read requests for *Region A* and *Region C* following the uniform distribution, while for *Region B zipf clustered* distribution. This time, the throughput of read operations was fixed at 1,000 ops/s and we also varied the access probability

of each *region*. The results are depicted in Figure 5.7(b), and once more the estimated server usage is very similar to the observed one, for all different access probabilities, ranging from 68%(when all *regions* compete equally for the cache) to 95% cache hit ratio.

Read and update mix experiments: By using both the read and the update model, we are able to estimate the server usage for read operations and update operations independently. However, as described in Section 5.4 the write path and the read path are mostly separated, thus we are able to estimate the overall server usage just by adding both estimations. In order to validate this assumption, we set up an experiment configured with different read and update mixes, namely: 90% read and 10% update; 80% read and 20% update; 50% read and 50% update; 20% read and 80% update; 10% read and 90% update. The *region* was populated with 4,000,000 records (4.3 GB) and the requests followed the uniform distribution with a fixed throughput of 118 ops/s, 562 ops/s, 1250 ops/s, 1958 ops/s and 2921 ops/s. These tested throughputs correspond to 5%, 20%, 40%, 60% and 80% server usage levels, as generated by the server model for the uniform distribution. In Figure 5.7 is depicted the results for this experiment. It shows that our approach is valid and it accurately predicts the server usage even when there are read and update operations simultaneously. However, as seen in Figure 5.6(c) and Figure 5.6(d) for the higher values of throughput the observed server usage is higher than the estimated one. These differences can be explained by compactions occurring during the test period that disrupt the readers of records stored on disk. Figure 5.8 shows the server usage along the entire 30 minute run for the 20% read and 80% update mix (Figure 5.6(d)) for the 2912 ops/s throughput. Until the compaction process starts (at 1277 seconds) the observed server usage average is the same as the estimated one (44%). Then, the compaction process greatly increases server usage to levels near 100%. When compaction ends regular behavior is resumed. This process greatly impacts the overall server usage average, but even at this point our estimated server usage is only off by 12%, which is the greatest difference observed. It is worth noting, however, that while more powerful hardware and particularly SSDs would attenuate the problem and help improve the estimation, in [Ahmad and Kemme 2015] it is proposed to offload compactions to a dedicated compaction server to prevent the significantly degraded read performance during compactions.

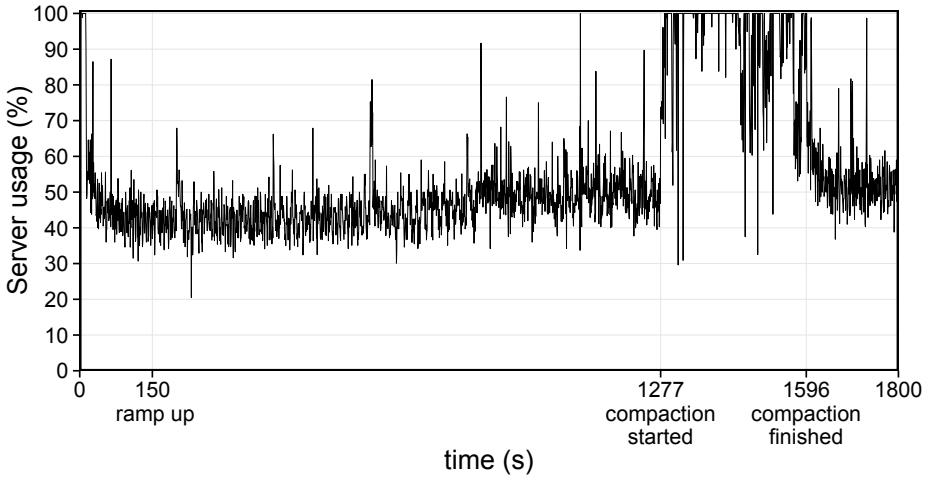


Figure 5.8: Observed server usage along a 30 minute run for the 20% read and 80% update mix for 2912 ops/s throughput.

Different hardware specification

All the previous experiments, including those in Section 5.2, were run using the same hardware specification. As a result, in this step of validation we want to show that the proposed approach is applicable to different hardware specification.

Configuration: In the following experiment we used two machines each with an Intel Xeon 2.3 GHz with 16 cores, with 24GB of main memory and a local SSD drive. HBase 0.98.3 was deployed, with one node acting as master for both HBase and HDFS, while holding a Zookeeper instance running in standalone mode. There was one *RegionServer* in another machine, configured with a heap of 12 GB, and co-located with a *DataNode*. HBase's LRU block cache was configured to use 55% of the heap size (6.6GB). YCSB was used as the workload generator, with the client running in the other node. All experiments were set to run for 30 minutes with 150 seconds of ramp-up time, and the server usage was logged every second in the server machine, using the UNIX top command. For every data point there were 3 independent runs, and the results presented are the computed average.

Single *region*: After building the server model based on the uniform distribution for the new hardware specification, we populated the HBase instance

using the YCSB's client with 18,000,000 records (20.5 GB). Again, the data size is larger than the available main memory to ensure that much of the data has to be brought from the SSD drive into main memory causing page in and page out. YCSB's client was configured to use the *zipf scrambled* distribution with 100% read operations. Similarly to previous experiments, the configured read throughput started at 250 ops/s and we stopped at 17,000 ops/s, which is when the server usage reached 80%. The results for this experiment are depicted in Figure 5.9. It shows the estimated server usage compared to the observed one. The estimated results are drawn from our approach using the server model for read operations, and observing the cache hit ratio as provided by HBase, which for this experiment is 59%. Again, the estimated server usage is very similar to the observed counterpart.

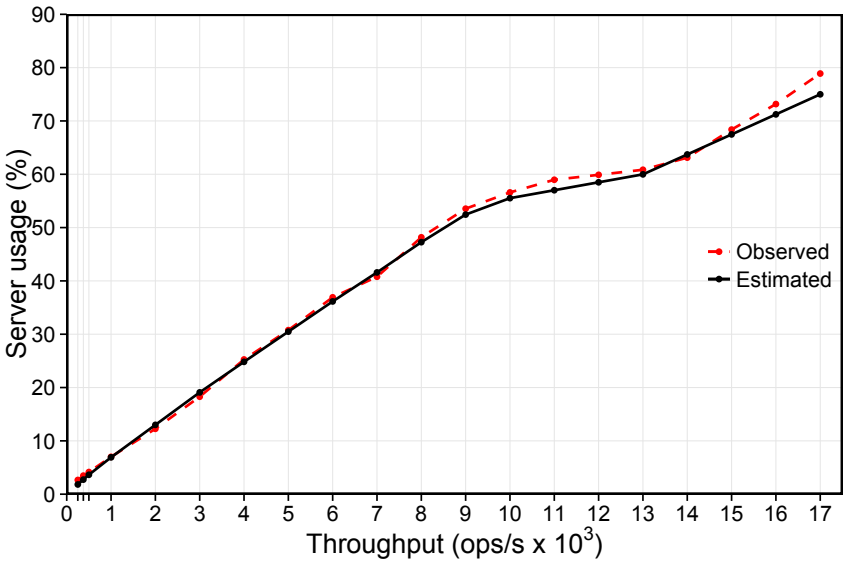


Figure 5.9: Experiment with 18 million records and *zip scrambed* distribution for a different hardware specification using HBase.

5.5.2 Cassandra

Configuration: In the experiments below we again used the same setting and hardware specification as in the experiments of Section 5.2. We used Cassandra version 2.1.7 in single node configuration, with a row cache size of 2GB, equivalent

to HBase. YCSB was used as workload generator, with the client running in other node. All experiments were set to run for 30 minutes with 150 seconds of ramp up time, and the server usage was logged every second in the *Cassandra* machine, using UNIX top command. For every data point there were 3 independent runs, and the results shown are the computed average.

Read experiments: The read experiments in Cassandra are analogous to the HBase single region experiments. That is, we wanted to assess if the proposed mechanisms can accurately predict the server usage when using other distributions than the uniform distribution. Consequently, we populated the Cassandra instance using the YCSB's client with 4,000,000 records (4.3GB) for the first experiment and with 3,000,000 records for the second experiment. Likewise, in the first experiment, the YCSB's client was configured to use *zipf clustered* distribution with 100% operations, and for a fixed throughput ranging from 250 ops/s to 10,000 ops/s. However, in the second experiment, instead of using the *zipf scrambled* distribution we had to use a *hotspot* distribution. Because, as Cassandra uses consistent hashing and its cache is row oriented instead of block oriented, the *zipf clustered* and the *zipf scrambled* distributions have the same cache hit ratio. Therefore, we used the *hotspot* distribution of Section 5.2, which for 3,000,000 records (3.2 GB) achieve the same cache hit ratio as the *zipf scrambled* distribution. The configured read throughput also ranged from 250 ops/s to 7,000 ops/s. The results for each experiment are depicted in Figure 5.9 and in Figure 5.5(d). They show the estimated server usage compared to the observed one. The estimated results are drawn from our approach using the generated model for read operations, and observing the cache hit ratio as provided by Cassandra exported metrics. Once again, the estimated server usage in both experiments is very similar to the observed one.

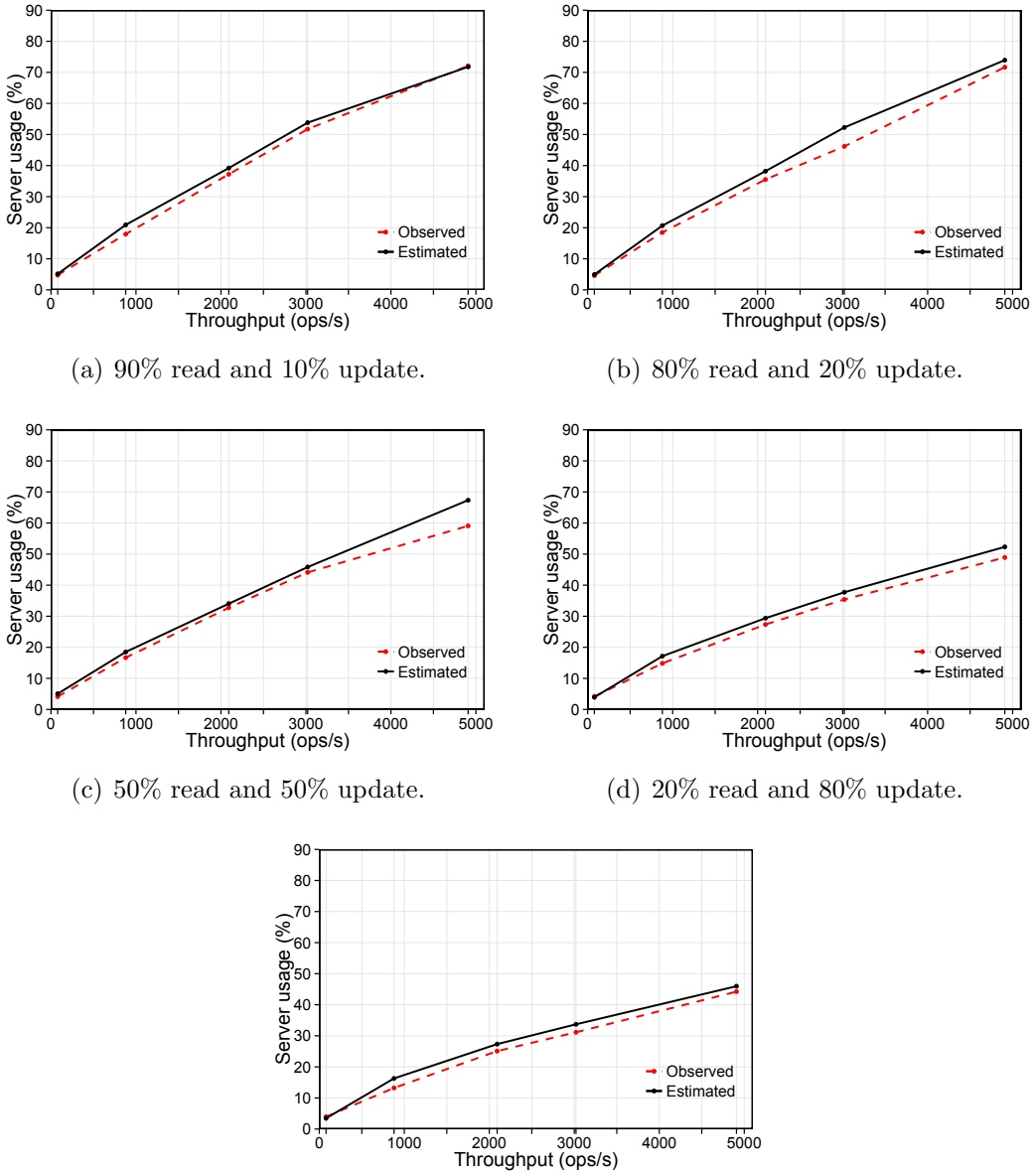
Read and update mix experiment: The write path of Cassandra is very similar to the write path of HBase. As a result, we should also be able to estimate the server usage when there is a mix of read and update operations. By using both the read and the update model, we are able to estimate the server usage for read operations and then sum the usage for update operations. Like HBase, we set up an experiment using the same mix of read and update operations, namely: 90% read and 10% update; 80% read and 20% update; 50% read

and 50% update; 20% read and 80% update; 10% read and 90% update. The *column family* was populated with 4,000,000, records (4.3 GB) and the requests followed the uniform distribution with a fixed throughput of 79 ops/s, 879 ops/s, 2092 ops/s, 3020 ops/s and 4908 ops/s. As shown in Figure 5.10 our approach is generic and can accurately predict the server usage when there are read and update operations simultaneously, even when using a different datastore. Contrary to HBase experiments the compaction process has not the same impact on the observed server usage.

From the above experiments we can state that our approach is valid, generic and can accurately estimate the server usage in HBase and Cassandra, regardless of the data size, of the request distribution, different hardware, multi-tenancy, and even when there is a mix of read and update operations. In fact, from all experiments the average accuracy is 94% with a standard deviation of just 5.6.

5.6 Discussion

Along this chapter we focused on a mechanism for NoSQL databases resource usage estimation. Our mechanism is able to accurately predict the resource utilization for every data size, request distribution and throughput combination. In contrast with previous approaches on prediction systems for cloud environments, we take advantage of focusing on a specific cloud component to improve prediction accuracy and its applicability. In particular, we observed that the majority of the NoSQL databases make use of buffer caching mechanisms to improve performance. Moreover, the effectiveness of such mechanisms is directly related to the performance and, as a consequence, to the resource utilization of the database. This effectiveness can be measured in terms of the hit ratio that the caching mechanism exhibits. The higher the cache hit ratio the more effective the cache mechanism is, and thus more performant is the database. In this work, we propose that instead of a specific workload is characterized by the three common parameters, namely: i) data size; ii) distribution of requests and iii) incoming throughput; a workload can be characterized by the incoming throughput and by its cache hit ratio, as the latter is a reflection of the i) data size and of the ii) distribution of requests. By making this simplification we can use the cache hit ratio and the throughput to build a server usage model based on the *uniform*



(a) 90% read and 10% update.

(b) 80% read and 20% update.

(c) 50% read and 50% update.

(d) 20% read and 80% update.

(e) 10% read and 90% update.

Figure 5.10: Read and update operations mix experiments in Cassandra.

distribution of requests, that can then be used to predict the resource utilization of any workload only by knowing those two parameters. In fact, in our experiments the average prediction accuracy achieved is 94%. Also, the solution does not require system traces or runtime mechanisms to improve the precision of the estimation. In addition, our approach is generic as it can be applied to different NoSQL databases, such as HBase and Cassandra.

Chapter 6

Resource Usage Load Balancer

Fully using the available resources is the ultimate goal for any application. This becomes even more important in a cloud environment dominated by the pay-as-you-go model where enterprises pay for the resources they consume. As a result, it is extremely important that, at all times, a particular application only consumes the resources that it really needs to achieve the desired performance. In the context of NoSQL databases due to its inherently distributed architecture, it is highly desirable that all the nodes that make up the cluster are to be fully utilized and the load is evenly distributed across the cluster.

In this chapter, we leverage the work described in the previous chapter to devise a new load balancer based on the estimation of the resource usage of NoSQL databases to maximize resource usage in each individual node, while minimizing the total number of nodes needed. This load balancer can be integrated into MeT framework to supersede the existing load balancer. As described in Chapter 3, MeT system provides a load balancing mechanism that tries to even the load across the NoSQL cluster. It does so by minimizing the number of requests per second (i.e the requested throughput) in each node. However, as we have shown in the previous chapter, due to the request distribution this approach may not minimize resource utilization across the cluster. In other words, if one node in the cluster is receiving 10,000 reads per second with a high cache hit ratio, while other node is getting the same 10,000 reads per second but with a much lower cache ratio, then although they have even load in terms of requests, the resource consumption will be very different in each node. As result, a load balancer based on the resource usage can more evenly distribute the load across the

cluster while simultaneously reducing the number of nodes needed to meet the desired performance.

In the following, we present the design, implementation and integration into MeT system of a load balancer mechanism based on the predicted resource usage. As our experiments show, the new load balancer achieves even better results than the original MeT's load balancer by achieving the same level of throughput while it maximizes resource usage in each individual node and, thus minimizing the total number of nodes needed.

6.1 Predicting cache hit ratio

In order to build a resource usage based load balancer we can not rely on the observed cache hit ratio to estimate the use of resources. As the load balancer distributes regions in an online fashion we first need to be able to accurately predict the cache hit ratio when one or more regions are to be placed in one node. Then, we can use that expected cache hit ratio in conjunction with the mechanisms described in the previous chapter to predict the resource usage in each node.

Calculating the cache hit ratio was, until recently, hard to accomplish for large cache sizes and large data sizes. However, Che, Tung and Wang, in what has become known as the *Che's approximation* [Che et al. 2002], proposed a simple approach to estimate the cache hit rate probability. The approximation is very accurate and versatile for a wide variety of popularity distributions and population sizes. The approximation operates under the LRU replacement algorithm and also under the independent reference model (IRM), which assumes that all requests are independent random variables with a common probability distribution. The *Che's approximation* then states that the hit rate, $hit(i)$, for a object i is approximated by:

$$hit(i) \approx 1 - e^{-p(i)tc}$$

where $p(i)$ is the probability of object i being requested, and tc is the *cache characteristic time*. The *cache characteristic time* can be defined as the time elapsed since an object i was last requested, such that i is no longer in the cache. For large cache sizes, a first approximation is to consider tc nearly deterministic,

and a second approximation is to consider *tc* a constant and thus independent of the specific object being requested. This greatly reduces the computational cost while still providing very accurate results. In fact, it is actually exact if the popularity distribution is uniform (the case where the hit rate is minimum for a LRU cache), and has a very low error when the popularity distribution is highly skewed (*Zipf* for instance). The average hit rate of a cache can then be calculated by:

$$p_{hit} = \sum_{i} p(i)hit(i)$$

In order to predict the cache hit ratio of a given data set, we need to know the probability of each object being requested. In HBase's LRU block cache, records are mapped to blocks, thus when one record is read the entire block it belongs to is brought into the block cache. Consequently, the probability of a object being requested does not refer to the individual record, but to the probability of a block being requested. We modified HBase server code to include a histogram that counts the number of times each block is accessed i.e. its access frequency. The histogram is maintained in a memory structure that is persisted periodically in HDFS. When a new *region* is created or moved to another *RegionServer* it is created a new instance of the histogram. Then, in order to be able to immediately access the histogram data its persistence period begins at 1 second and increases in a quadratic manner until it reaches 30 seconds. Thereafter the persistency period is maintained at 30 seconds so the overhead is kept to a minimum. When the *region* is closed or moved to other RegionServer this process is repeated. It is worth noting that HBase already exports a metric that counts the number of times each block is accessed, but reduces it to a single global metric. As a result, we just distinguish between the accessed blocks, which results in almost no overhead when compared to the vanilla HBase.

From the histogram we can compute the *probability mass function* (PMF) needed to calculate the cache hit ratio. However, there is a small detail. When trying to calculate the cache hit ratio of two or more independent *regions* we get two or more histograms, corresponding to the different *regions*. In this case, we have to merge the histograms into a single one. This can be done because the cache is a shared resource among all *regions* served by the *RegionServer*.

Implementation

We first used an implementation of the *che's approximation* from an open-source Python project called Icarus [Saino et al. 2014]. However, from our experiments the Python mathematical libraries used by Icarus (namely, scipy and numpy) resulted in very slow times to calculate the *cache characteristic time* for our cache sizes and data sizes. In fact, those times were in the order of several minutes, and therefore incompatible with an online load balancer. As a result, we have developed our own implementation of the *che's approximation* resorting to R software with an interface to Python. This allowed us to lower the computational times down to just a few seconds.

We have conducted several experiments to assert that our implementation's calculated cache hit ratio was similar to the observed cache hit ratio as provided by HBase metrics. In fact, from our experiments we can state that the results are as expected with the calculated cache hit ratio very close to the observed cache hit ratio with a deviation of less than 1%.

The process described allows to accurately predict the cache hit ratio for any number of *regions*, regardless of the their request distribution or cache size.

6.2 Algorithm and MeT integration

As described in Chapter 3, MeT provides a load balancing mechanism that tries to even the load across the NoSQL cluster. It does so by minimizing the number of requests per second (i.e the requested throughput) in each node. Briefly, this load balancing mechanism (Algorithm 2) assigns data partitions to nodes using a *makespan minimization* algorithm, which encompasses two main stages. In the first stage, all data partitions are first sorted by decreasing order according to the observed requested throughput. In the second stage, in decreasing order each data partition is assigned to the least loaded node until there are no more data partitions left.

The resource usage load balancer follows the exact same algorithm to assign data partitions to nodes. But, instead of minimizing the number of requests in each node, it will minimize the resource usage in each node. In order to do so, we need to reimplement two functions, namely: i) the sorting function and the ii) the computation of the least loaded node. In Algorithm 2 they correspond to

the *dataPartitions.sort()* and *nodeGroup.getMostEmptyNode()* functions, respectively.

The new sorting function has to sort *regions* by decreasing order of predicted resource usage. As a result, we must first estimate the resource usage of each *region*. This resource usage corresponds to the consumption each region would have if it were placed alone in the *RegionServer*. Consequently, the sorting function is composed of four main steps:

1. Download from HDFS the histograms of all *regions*;
2. Calculate the cache hit ratio of each *region* using our implementation of the *che's approximation*;
3. Estimate the resource usage of each *region* taking the predicted cache hit ratio and the observed metrics regarding the number of requests, which are then fed as inputs to the mechanisms described in the previous chapter;
4. Sort *regions* by decreasing order of predicted resource usage.

In the case of the least loaded node function we have to assign each *region* to the node with the least predicted resource usage until there are no *regions* left. As opposed to the sorting function, we have to take into account all *regions* that are to be placed in the node so we need to merge the histograms to calculate the global cache hit ratio and thereafter the predicted resource usage. The least loaded node function has two main steps:

1. Take the histogram of the *region* being assigned and merge it with the histograms of the *regions* that were previously assigned to the node, in order to calculate the global cache hit ratio using our implementation of the *che's approximation*;
2. Estimate the resource usage for the node, taking the global predicted cache hit ratio and the observed metrics regarding the number of requests of each assigned *regions*, which are then fed as inputs to the mechanisms described in the previous chapter.

Integrating the resource usage load balancer into MeT is a straightforward task after we have reimplemented the sorting and the least loaded node functions,

as well as all the auxiliary functions needed. As a consequence, Python language was used for all functions and, within MeT we can easily choose which load balancer mechanism to use.

6.3 Evaluation

In this section we evaluate the resource usage load balancer integrated into MeT and we compare it with MeT's original load balancer. For this purpose, we chose PyTPCC an optimized implementation for HBase of the standard OLTP benchmark TPC-C. Note that, while TPC-C standard transactions are expected to have full ACID semantics this implementation offers the isolation semantics provided by HBase: record level atomicity.

TPC-C benchmark attempts to reproduce the behavior of any business in which sales' districts are geographically distributed along with the corresponding warehouses. There are a total of 9 tables and 5 different types of transactions, and the results are measured in transactions per minute (tpmCs). The default traffic is a mixture of 8% read-only and 92% update transactions and thus is a write intensive benchmark.

The TPC-C database was populated with 60 warehouses resulting in a database of 13 GB. The experiment starts with a HBase cluster composed of 1 *RegionServer*, configured with a heap of 4 GB, and co-located with 1 *DataNode*. We used another machine as the master of both HBase and HDFS as well as the Zookeeper instance. PyTPCC's clients were deployed in two other machines amounting to 600 clients (300 client threads per machine). All nodes used for these experiments have an Intel i3 CPU at 3.1GHz, with 8 GB of memory and a local 7200 RPM SATA disk, and are interconnected by a switched Gigabit local area network.

MeT configuration

We used the same MeT configuration parameters as the experiments described in Chapter 3. Briefly, every 30 seconds the *Monitor* component gathers the metrics and sends them to the *Decision Maker* every 3 minutes. The *Decision Maker* is only invoked after having 6 samples to minimize the impact of sudden spikes and take advantage of the exponential smoothing algorithm. The HBase configuration

parameters for each group (*Distribution Algorithm* of Section 5.3) are described in Table 6.1, with cache size in percentage of heap size and the correspondent number of blocks, as well as the memstore size.

Node profile	Cache size (%)	Cache size (blocks)	Memstore size (%)
Read	55%	34350	10%
Write	10%	6245	55%
Read/Write	45%	28105	20%
Scan	55%	34350	10%

Table 6.1: Node configuration profiles.

Results

The following experiments are divided into two phases. In the first phase, the experiment starts with a HBase cluster composed of 1 *RegionServer* co-located with 1 *DataNode*. After a ramp-up period of 4 minutes MeT begins. Then, the *Monitor* component periodically collects the several performance metrics and after 3 minutes invokes the *Decision Maker*, which takes the decision of adding or removing *RegionServers/DataNodes* to the HBase cluster. In an open benchmark like PyTPCC the maximum throughput is achieved when the client PyTPCC machines are saturated. Therefore, the goal of this first phase is to know how many *RegionServers* are needed to saturate the 2 PyTPCC client machines (600 clients) under two different scenarios: i) MeT configured with the regular load balancer; and ii) MeT configured with the resource usage load balancer. In both scenarios, we let MeT configure the cluster and when MeT considers the cluster to be healthy and, thus it takes no further action, we stop both MeT and the PyTPCC benchmark clients. The first phase of the experiment under both scenarios is depicted in Figure 6.1. As can be observed, MeT configured with the resource usage load balancer settles down at 3 nodes, while MeT configured with the regular load balancer needs 4 nodes. In fact, MeT configured with the resource usage load balancer just needs 3135 seconds to achieve a stable cluster. MeT with the default load balancer needs more 779 seconds to achieve a healthy cluster. It is worth noting, however, that the regular load balancer also tries to manage the load with only 3 nodes, but from all of our experiments it fails to do so and, thus it always adds another *RegionServer* settling at 4 *RegionServers*.

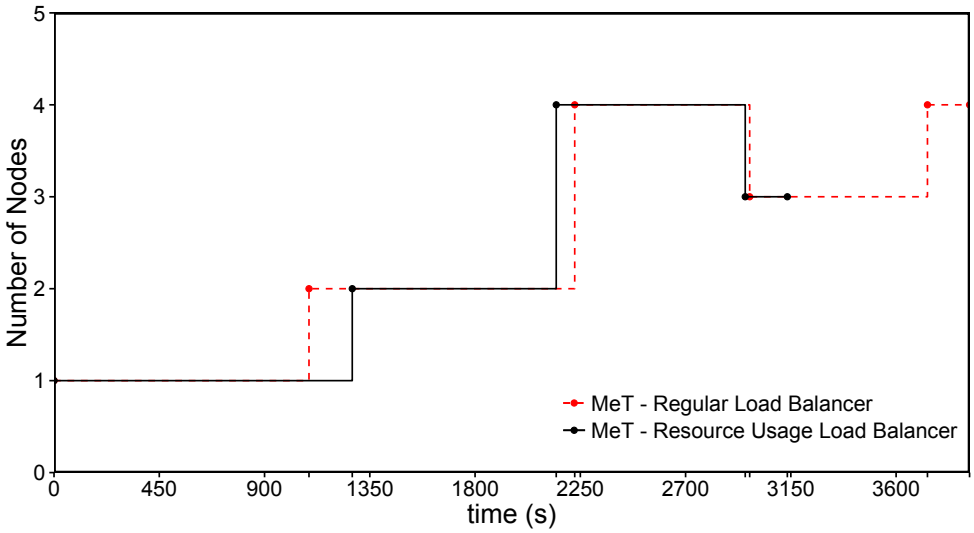


Figure 6.1: Maximum number of nodes needed to saturate the client PyTPCC machines.

In the second phase of the experiment, we wanted to measure the throughput in tpmCs of the configuration achieved for both scenarios. As a result, immediately after the cluster was healthy and we stopped both the MeT system and the PyTPCC clients, we restarted the PyTPCC clients and let the experiment run for 900 seconds. The whole process was repeated 3 times. In addition, we also measured the maximum throughput possible for the PyTPCC client machines. In this regard, we have added enough *RegionServers* to the cluster for a total of 7 so that manually distributing *regions* was easy and no node would be overloaded. The maximum throughput achieved was then 4224 tpmCs. Table 6.2 shows the average results for the two different load balancer configurations. As shown in the table both scenarios achieve the maximum throughput possible for the PyTPCC clients with around 4224 tpmCs. This is a very interesting result since the resource usage load balancer can saturate the PyTPCC clients with just 3 *RegionServers*. This must mean that the resource usage in the cluster is higher, and the load is more evenly distributed across the cluster.

Scenario	Throughput (tpmCs)	% Max Throughput	#Nodes
Regular Load Balancer	4222.66	100	4
Resource Usage Load Balancer	4223.86	100	3

Table 6.2: PyTPCC average throughput results.

In order to show that this is indeed the case we have measured the server usage in each of the *RegionServer* machines. The results are depicted in Table 6.3.

Scenario	RS 1 (%)	RS 2 (%)	RS 3 (%)	RS 4 (%)
Regular Load Balancer	76.22	48.35	69.25	4.47
Resource Usage Load Balancer	68.88	59.19	60.00	None

Table 6.3: Average server usage for each *RegionServer* machine during the PyTPCC experiment.

As can be observed, MeT configured with the resource usage load balancer can more evenly distribute the load across all *RegionServers* with very similar average server usage in each server, and thus allowing it achieve the maximum throughput with just 3 *RegionServers*. On the other hand, MeT configured with the regular load balancer needs one more *RegionServer* to achieve the maximum throughput but the load is not evenly balance. *RegionServer* 1 is under heavy utilization, while *RegionServer* 4 was underutilized.

From these experiments, we can conclude that the resource usage load balancer makes better use of available resources. It achieves the same level of throughput of the regular load balancer, but it minimizes the total number of nodes required.

6.4 Discussion

In this chapter, we focused on the design and implementation of a load balancer for NoSQL databases based on the predicted resource usage. The new load balancer leverages the mechanisms described in the previous chapter regarding the estimation of resource usage, to optimize the server usage on each individual node, and thus minimizing the number of nodes required to meet the performance goals. However, as an online load balancer it can not rely on the observation cache hit ratio to calculate the resource usage. Instead, it must be able to predict the cache hit ratio when one or more *regions* are placed in a *RegionServer*. In order to predict the cache hit ratio of a given data set we resorted to the *che's approximation*. The approximation is very accurate and versatile for a wide variety of popularity distributions and population sizes, and operates under the LRU cache replacement algorithm. Although there are some research projects that already

implement the approximation, our tests revealed that none of the implementations would offer the performance needed for an online load balancer. Therefore, we have implemented the approximation using R software with an interface to Python. This allowed us to lower the computational times down to just a few seconds, and from our experiments the predicted cache hit ratio is very close to the observed cache hit ratio with a deviation of less than 1%.

One of the requirements of the approximation is that we know the probability of each object being requested. Consequently, we have modified HBase server code to include for every *region* a histogram of the access frequency to each block. The histogram is periodically persisted in HDFS, and imposes almost no overhead to HBase. From the histogram we can compute the *probability mass function* (PMF) needed to calculate the cache hit ratio under *che's approximation.*

In order to integrate the new load balancer into MeT system load balancing algorithm, we needed to reimplement two functions as well as all the auxiliary functions. This gives MeT the flexibility to easily switch between both load balancer implementations. As our experiments show, the new load balancer achieves even better results than the original MeT 's load balancer. It attains the same level of throughput while it maximizes the resource usage on each individual node, and it also minimizes the total number of nodes required.

Chapter 7

Conclusion

Cloud Computing is the current trend in systems design and conception. The *cloud* is a complex environment composed of various subsystems that, although different, are expected to exhibit a set of fundamental features: high availability, high performance and elasticity. While high availability and high performance are common goals to all systems, elasticity is specific to the *cloud* environment and closely tied to the pay-as-you go model. Elasticity can be defined as the ability of a system to grow or shrink its resource consumption according to demand. The ability to adjust resource consumption according to demand, favors the pay-as-you-go model and improves resource utilization. In addition, current *cloud* providers make available their infrastructure (IaaS), platform (PaaS) or software (SaaS) to multiple customers in a multi-tenant environment. Therefore, optimal resource utilization becomes an even greater concern, since if one customer is using more resources than needed, it may impact the performance of other customer's applications, resulting in poorer overall performance. From a *cloud* provider's perspective, the ability to dynamically optimize resource usage according to the contracted level of service is fundamental to the business model.

Along this dissertation work we focused on a specific component: cloud-based data stores, popularly referred to as NoSQL databases. These databases have been designed to take advantage of large resource pools and provide high availability and high performance. Moreover, they were designed to cope with resource availability changes. For instance, it is possible to add or remove database nodes from the cluster and to have the database handle such change transparently. However, even though NoSQL databases can handle elasticity, they are not au-

tonomously elastic: an external entity is required to control when and how to add or remove nodes. Therefore, we designed and implemented MeT framework that provides automated elasticity to NoSQL databases. MeT not only adds and removes nodes, but also introduces a novel concept to heterogeneously configure performance related parameters of NoSQL systems, according to the observed workloads. We achieve this by leveraging on an existing IaaS system as the basic provider of elasticity. We expose new database engine metrics regarding workload's access patterns, which are constantly monitored along with the IaaS nodes. This information feeds our decision component that then performs online cluster reconfiguration as needed. As our experiments show, fine tuning the available parameters of a NoSQL database on a per node basis, significantly boosts overall performance, specially when considering the workload characteristics. Furthermore, when comparing MeT with state-of-art systems, MeT was able to clearly achieve better performance while using less resources.

Following the work on MeT, some interesting research paths were identified. We focus on two. Firstly, NoSQL databases assume data partitioning and as applications have different access patterns there may exist data hotspots. Identifying and finding a correct splitting point to eliminate a hotspot is critical to the optimal performance of NoSQL databases. In this work we presented a workload aware data partitioning algorithm for NoSQL databases. The algorithm is based on the online median estimation of past requests in order to find the ideal splitting point. The results obtained showed the mechanism is effective both for achieving good load balance as well as improving overall performance of the NoSQL database, and is a great complement to MeT framework.

Secondly, we focused on a mechanism for NoSQL databases resource usage prediction. By observing that the majority of NoSQL databases make use of buffer caching to improve performance we were able to draw a relationship correlating the cache hit ratio with the resource usage. Furthermore, we propose that instead of a specific workload is characterized by the three common parameters, namely: i) data size; ii) distribution of requests and iii) incoming throughput; a workload can be characterized by the incoming throughput and by its cache hit ratio, as the latter is a reflection of the i) data size and of the ii) distribution of requests. Building on this simplification we can use the cache hit ratio and the throughput to build a server usage model based on the uniform distribution

of requests, that can then be used to estimate the resource utilization of any workload only by knowing those two parameters. The approach is generic as it can be applied to different NoSQL databases, such as HBase and Cassandra.

Finally, we leveraged the work on resource usage estimation to devise a new load balancer that was integrated into MeT framework. While MeT's original load balancer tries to even the load across the NoSQL cluster by minimizing the number of requests per second in each node, the resource usage load balancer can more evenly distribute the load across the cluster, while simultaneously reducing the number of nodes needed to meet the desired performance. As our experiments show, the resource usage load balancer achieves even better results than the original MeT 's load balancer. It attains the same level of throughput while maximizing the resource usage in each individual node and, thus minimizing the total number of nodes required.

Bibliography

Muhammad Yousuf Ahmad and Bettina Kemme. Compaction management in distributed key-value datastores. *Proc. VLDB Endow.*, 8(8):850–861, April 2015. - **Cited** on page 86.

Apache. Hadoop: `http://hadoop.apache.org/` (2015). - **Cited** on page 10.

Google AppEngine. Google app engine documentation. `http://code.google.com/appengine/docs/`. accessed 19th January, 2016. - **Cited** on page 2.

M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 2010. - **Cited** on pages 2 and 15.

Microsoft Azure. Microsoft azure. `http://www.microsoft.com/windowsazure`. accessed 19th January, 2016. - **Cited** on page 2.

R. G. Brown. *Smoothing, forecasting and prediction of discrete time series.* Prentice-Hall, 1963. - **Cited** on page 32.

Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association. - **Cited** on page 10.

F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI*, 2006. - **Cited** on pages 3 and 9.

Hao Che, Ye Tung, and Zhijun Wang. Hierarchical web caching systems: modeling, design and experimental results. *IEEE Journal on Selected Areas in Communications*, 20(7):1305–1314, 2002. - **Cited** on pages 63, 69 and 94.

S. Chen, A. Ailamaki, M. Athanassoulis, P. B. Gibbons, R. Johnson, I. Pandis, and R. Stoica. Tpc-e vs. tpc-c: characterizing the new tpc-e benchmark via an i/o comparison study. *SIGMOD Record*, 2010. - **Cited** on page 23.

Amazon CloudWatch. Cloudwatch. `http://aws.amazon.com/cloudwatch/`. - **Cited** on pages 16 and 44.

B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *VLDB Endowment*, 2008. - **Cited** on pages 3 and 9.

B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010. - **Cited** on pages 23, 56 and 64.

C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. In *VLDB Endowment*, 2010. - **Cited** on page 17.

Barthélémy Dagenais. Py4j - a bridge between python and java. `http://py4j.sourceforge.net/`. - **Cited** on page 38.

G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, 2007. - **Cited** on pages 3 and 9.

Peter Desnoyers, Timothy Wood, Prashant Shenoy, Rahul Singh, Sangameshwar Patil, and Harrick Vin. Modellus: Automated modeling of complex internet data center applications. *ACM Trans. Web*, pages 1–29, 2012. - **Cited** on page 19.

Diego Didona, Francesco Quaglia, Paolo Romano, and Ennio Torre. Enhancing performance prediction robustness by combining analytical modeling and machine learning. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 145–156, 2015. - **Cited** on page 19.

Amazon EC2. Amazon ec2 documentation. `http://aws.amazon.com/ec2/`. accessed 19th January, 2016. - **Cited** on page 2.

R. A. Fisher. On the probable error of a coefficient of correlation deduced from a small sample. *Metron*, pages 3–32, 1921. - **Cited** on page 67.

L. George. *HBase: The Definitive Guide*. O'Reilly, 2011. - **Cited** on pages 3 and 9.

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. - **Cited** on page 10.

Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002. - **Cited** on page 12.

Zhenhuan Gong, Xiaohui Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *International Conference on Network and Service Management*, pages 9–16, 2010. - **Cited** on page 19.

R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 1969. - **Cited** on page 35.

Naohiro Hayashibara, Xavier Defago, Rami Yared, and Takuya Katayama. The ϕ Accrual Failure Detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, SRDS '04, pages 66–78, Washington, DC, USA, 2004. IEEE Computer Society. - **Cited** on page 14.

Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, 2010. - **Cited** on pages 10 and 64.

Akshay Java, Xiaodan Song, Tim Finin, and Belle Tseng. Why we twitter: understanding microblogging usage and communities. In *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis*, pages 56–65, New York, NY, USA, 2007. - **Cited** on page 51.

Brendan Jennings and Rolf Stadler. Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management*, pages 1–53, 2014. - **Cited** on page 18.

David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOCK*, 1997. - **Cited** on page 12.

A. Khan, X. Yan, Shu Tao, and N. Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *Network Operations and Management Symposium (NOMS)*, pages 1287–1294, 2012. - **Cited** on page 19.

Markus Klems, Adam Silberstein, Jianjun Chen, Masood Mortazavi, Sahaya Andrews Albert, P.P.S. Narayan, Adwait Tumbde, and Brian Cooper. The yahoo!: Cloud datastore load balancer. In *Proceedings of the Fourth International Workshop on Cloud Data Management*, CloudDB '12, pages 33–40, 2012. - **Cited** on pages 17 and 19.

I. Konstantinou, E. Angelou, D. Tsoumakos, C. Boumpouka, N. Koziris, and S. Sioutas. Tiramola: Elastic nosql provisioning through a cloud management platform. In *International Conference on Management of Data (SIGMOD Demo Track)*, 2012. - **Cited** on pages 16, 19, 40, 44 and 72.

Avinash L. and Prashant M. Cassandra - a decentralized structured storage system. In *LADIS*, 2009. - **Cited** on pages 3 and 9.

J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. In *Studies in integer programming (Proc. Workshop, Bonn, 1975)*. North-Holland, 1977. - **Cited** on page 35.

Jiexing Li, Arnd Christian König, Vivek Narasayya, and Surajit Chaudhuri. Robust estimation of resource consumption for sql queries using statistical techniques. *Proc. VLDB Endow.*, 5(11):1555–1566, 2012. - **Cited** on page 19.

H.C. Lim, S Babu, and J.S. Chase. Automated control for elastic storage. *7th international conference on Autonomic computing*, 2010. - **Cited** on pages 15 and 16.

M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: Design, implementation and experience. *Parallel Computing*, 2003. - **Cited** on page 38.

Peter M. Mell and Timothy Grance. Sp 800-145. the nist definition of cloud computing. Technical report, Gaithersburg, MD, United States, 2011. - **Cited** on page 15.

Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. Performance and resource modeling in highly-concurrent oltp workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 301–312, 2013a. - **Cited** on page 18.

Barzan Mozafari, Carlo Curino, and Samuel Madden. Dbseer: Resource and performance prediction for building a next generation database cloud. In *Conference on Innovative Data Systems Research (CIDR)*, 2013b. - **Cited** on pages 18 and 19.

OpenStack Foundation. Openstack blog entry: 'openstack foundation update'. `http://www.openstack.org/blog/2012/04/openstack-foundation-update/`. [Online; last accessed December-2015]. - **Cited** on pages 4 and 38.

D. Owens. Securing elasticity in the cloud. *Communications of the ACM*, 2010. URL `http://doi.acm.org/10.1145/1743546.1743565`. - **Cited** on page 1.

A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *ACM SIGMOD International Conference on Management of Data*, 2012. - **Cited** on page 17.

Amazon S3. Amazon s3 documentation. `http://aws.amazon.com/s3/`. accessed 19th January, 2016. - **Cited** on page 2.

Lorenzo Saino, Ioannis Psaras, and George Pavlou. Icarus: A caching simulator for information centric networking (icn). In *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*, pages 66–75, 2014. - **Cited** on page 96.

Amazon Auto Scaling. Auto scaling. `http://aws.amazon.com/autoscaling/`. - **Cited** on pages 16 and 44.

Rahul Singh, Upendra Sharma, Emmanuel Cecchet, and Prashant Shenoy. Autonomic mix-aware provisioning for non-stationary data center workloads. In *Proceedings of the 7th International Conference on Autonomic Computing*, pages 21–30, 2010. - **Cited** on page 19.

David B. Skillicorn. The Case for Datacentric Grids. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 313–, Washington, DC, USA, 2002. IEEE Computer Society. - **Cited** on page 2.

Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, pages 202–208, 1985. - **Cited** on page 13.

A.A. Soror, U.F. Minhas, A Aboulnaga, K Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. *ACM SIGMOD international conference on Management of data*, 2008. - **Cited** on page 17.

M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, 2007. - **Cited** on page 43.

S. Sudevalayam and P. Kulkarni. Affinity-aware modeling of cpu usage for provisioning virtualized applications. In *2011 IEEE International Conference on Cloud Computing (CLOUD)*, pages 139–146, 2011. - **Cited** on page 18.

A.L. Tatarowicz, C. Curino, E.P.C. Jones, and S. Madden. Lookup tables: Fine-grained partitioning for distributed databases. In *ICDE*, 2012. - **Cited** on page 17.

B. Trushkowsky, P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. The scads director: scaling a distributed storage system under stringent performance requirements. 2011a. - **Cited** on page 19.

B Trushkowsky, P. Bodík, A Fox, M.J. Franklin, M.I. Jordan, and D.A. Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. *FAST*, 2011b. - **Cited** on page 16.

L. M. Vaquero, L. Rodero-Merino, and R. Buyya. Dynamically scaling applications in the cloud. *ACM SIGCOMM*, 2011. - **Cited** on pages 1 and 15.

Ricardo Vilaça, Francisco Cruz, and Rui Oliveira. On the expressiveness and trade-offs of large scale tuple stores. In *Proceedings of the 2010 international conference on On the move to meaningful internet systems: Part II*, OTM'10, pages 727–744, Berlin, Heidelberg, 2010. Springer-Verlag. - **Cited** on page 3.

Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: Enabling high-level slos on shared storage systems. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, SoCC '12, pages 14:1–14:14, 2012. - **Cited** on page 19.

Timothy Wood, Ludmila Cherkasova, Kivanc Ozonat, and Prashant Shenoy. Profiling and modeling resource usage of virtualized applications. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 366–387, 2008. - **Cited** on page 18.

Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Proceedings of the 4th International Conference on Autonomic Computing*, pages 27–, 2007. - **Cited** on page 19.