Otto-von-Guericke University Magdeburg

Faculty of Computer Science

Dissertation

Self-tuning for Cloud Database Clusters

Author:

M.Sc. Siba Mohammad

April 22, 2016

Supervisor:

Prof. Dr. rer. nat. habil. Gunter Saake

Department of Technical and Business Information Systems

Mohammad, Siba:
Self-tuning for Cloud Database Clusters
Dissertation, Otto-von-Guericke University Magdeburg, 2016.

Self-tuning for Cloud Database Clusters

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieurin (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von M.Sc. Siba Mohammad

geb. am 07.02.1985 in Latakia, Syria

Gutachterinnen/Gutachter

Prof. Dr. Gunter Saake
Prof. Dr. Kai-Uwe Sattler
Prof. Dr. Andreas Thor

Magdeburg, den 22.04.2016

Abstract

The well known approaches of tuning and self-tuning of data management systems are essential in the context of the Cloud environment, which promises self management properties, such as elasticity, scalability, and fault tolerance. Moreover, the intricate Cloud storage systems criteria, such as their modular, distributed, and multi-layered architecture, add to the complexity of the tuning process and necessity of the self-tuning process.

Furthermore, if we have one or more applications with one or more workloads with contradicting and possibly changing optimization goals, we are faced with the question of how to tune the underlying storage system cluster to best achieve the optimization goals of all workloads. Based on that, we define the tuning problem as finding the cluster configuration out of a set of possible configurations that would minimize the aggregated cost value for all workloads while still fulfilling their performance thresholds.

In order to solve such a problem, we investigate the design and implementation of a Cloud storage system agnostic (self-)tuning framework. This framework consists of components to observe, and model different performance criteria of the underlying Cloud storage system. It also includes a decision model to configure tuning parameters based on applications requirements. To model the performance of the underlying Cloud storage system, we use statistical machine learning techniques. The statistical data that is needed to model the performance can be generated in a training phase. For that we designed a training component that generates workloads and automates the testing process with different cluster configurations.

As part of our evaluation, we address the essential problem of tuning the cluster size of the Cloud storage system while minimizing the latency for the targeted workloads. In order to do that, we model the latency in relation to cluster size and workload characteristics. The predictive models can then be used by the decision component to search for the optimal allocation of nodes to workloads.

We also evaluate different alternatives for the search algorithms as part of the decision component implementation. These alternatives include brute-force, and genetic algorithm approaches.

Zusammenfassung

Die bekannten Ansätze zum Tuning und Self-Tuning von Datenverwaltungssystemen sind in Cloud-Umgebungen, welche Selbstverwaltungseigenschaften wie Elastizität, Skalierbarkeit und Ausfallsicherheit versprechen, von essentieller Bedeutung. Darüber hinaus erhöhen die komplexen Eigenschaften von Cloud Storage-Systemen, wie zum Beispiel ihre modulare, verteilte und mehrschichtigen Architektur, die Unübersichtlichkeit des Tuning-Vorgangs und damit die Notwendigkeit eines Self-Tuning-Prozesses.

Weiterhin stellt sich durch verschiedene Anwendungen mit unterschiedlichen Workloads und sich widersprechenden und womöglich veränderlichen Optimierungszielen die Frage, wie der gemeinsame Cloud Storage Cluster eingerichtet werden kann, um die Optimierungsziele aller Workloads zu erreichen. Ausgehend von dieser Fragestellung definieren wir das vorliegende Tuning-Problem als die Suche nach einer Cluster-Konfiguration aus einer Menge möglicher Konfigurationen, welche die aggregierten Kosten aller Workloads minimiert und dabei ihre Leistungsanforderungen erfüllt.

Um dieses Problem zu lösen, untersuchen wir die Möglichkeiten zum Entwurf und zur Implementierung eines Self-tuning Frameworks, welches unabhängig vom konkret verwendeten Cloud Storage System ist. Dieses Framework besteht aus Komponenten zur Überwachung und Modellierung verschiedener Performance-Kriterien des Cloud Storage-Systemen. Es beinhaltet ebenfalls ein Entscheidungsmodell, mit dem Tuning-Parameter basierend auf Anwendungsanforderungen konfiguriert werden können. Zur Modellierung der Performance des Cloud Storage-Systemen verwenden wir Techniken des maschinellen aschinellen Lernens. Die statistischen Daten, welche zur Ableitung des Performance-Modells notwendig sind, können in einer Trainingsphase generiert werden. Hierzu haben wir eine Trainingskomponente entwickelt, welche verschiedene Workloads generiert und den Test mit verschiedenen Cluster-Konfigurationen automatisiert.

Als Teil unserer Evaluation untersuchen wir das grundlegende Problem des Konfiguioerens der Cluster-Größe zur Miminierung der Latenz der auszuführenden Workloads. Um dies zu erreichen, modellieren wir die Latenz in Abhängigkeit von der Cluster-Größe und spezifischen Workload-Eigenschaften. Die Vorhersagemodelle können dann von der Entscheidungskomponente genutzt werden, um die optimale Allokation von Knoten zu Workloads zu ermitteln. Wir evaluieren ebenfalls verschiedene Alternativen für Suchalgorithmen als Teil der Entscheidungskomponente. Die Alternativen sind hierbei eine Brute-Force-Methode sowie ein genetischer Algorithmus.

Acknowledgments

This work would not have been possible without the support of many people. First, I would like to thank my supervisor Prof. Gunter Saake for giving me the chance to write this thesis as a member of the DBSE group. Prof. Saake gave me the freedom to pursue my research direction, and provided me guidance and support. Even with his busy schedule, he manages to find time, whenever one approaches him for advice or discussion.

Many thanks to my colleague Dr.-Ing Eike Schallehn. He helped me start at the DBSE group, and supported me during the different phases of my work. While we shared the same office, I have learnt a lot from him as an academic and a person. Thanks for the many brainstorming sessions, helpful feedback, and motivation.

I would like to thank my family for showing me unconditional love and support. I want to thank my mother for being an example of hard working career woman and devoted parent. I want to thank my father for handing me books, as answers to my questions, since I was a little girl. The “Yes, you can!” attitude from both of them lead me to where I am today. I would like to thank my little sister and her husband for always being by my side, bringing positive vibes and humor through tough times.

Special thanks to Annette and Elmar Dahmen-Eisenberg. Thank you for the love, kindness, and parental attitude that you showed me since I came to Germany. Thanks for making me feel home.

I would like to thank my dear friends Diana Nikolaus, Kirsten Bröcker, Hala Ismael, Saade Saad, Naoum Jamous, Razan Issa, Lugain Khalifa, and Muhannad Ali. They helped me cope with deadlines stress, rejected papers, and failed experiments. When things got better and my papers were accepted, they made sure that I celebrate to the maximum. Thank you guys for recharging my battery, so I could continue. Thank you for dragging me out of the office and providing much needed distraction. Thank you for being with me through the years, and making life such an interesting journey.

Last but not least, I would like to thank the current and former members of the DBSE group. I have learnt a lot during my work as a member in such a competitive, and diverse research group. Thank you all for your friendly attitude, and many teaching moments.

Contents

List of Figures **xiv**

List of Tables **xv**

List of Code Listings **xvii**

1 Introduction **1**
1.1 Goal of the Thesis 2
1.2 Outline 3

2 Background and Related Work **5**
2.1 Cloud Data Management 5
2.1.1 Cloud Environment 5
2.1.2 Cloud Requirements for Storage System 7
2.1.3 Relational Database Management Systems as a Service 8
2.1.4 Not only SQL Storage Systems 9
2.2 Database Tuning and Self-tuning 18
2.2.1 The General Process of System Tuning 18
2.2.2 Database Tuning 19
2.2.3 Database Self-tuning 21
2.2.4 (Self-)Tuning Cloud Data Management Systems 23
2.2.5 Related Work 25
2.3 Performance Modeling 26
2.3.1 Performance Modeling Approaches for Database Systems 27
2.3.2 Data-driven Modeling Approach 28
2.3.3 Related Work 29
2.4 Summary 34

3 (Self-)Tuning Cloud Storage Clusters **37**
3.1 Overview and Scope 37
3.2 A Framework for Self-tuning Cloud Storage Clusters 41
3.2.1 General Problem Statement 41
3.2.2 Clustering the Cloud 42
3.2.3 Framework Design 43
3.2.4 Training Component 45

3.2.5 Cost Estimation Component 51
3.2.6 Decision Component 53
3.2.7 Monitoring and Refinement Component 55
3.3 Framework Usage 55
3.3.1 Static/Offline Tuning during System Cluster Design 56
3.3.2 Dynamic/Online Tuning during System Cluster Deployment . . 57
3.3.3 Offline-Online Tuning Process 58
3.4 Summary 60

4 Prototype Implementation 63
4.1 Required Technologies 63
4.1.1 Implementation Environment 63
4.1.2 Cloud Storage System 64
4.2 Implementation 66
4.2.1 Deployment of the Storage Cluster 66
4.2.2 Workload Generation 73
4.2.3 Data Analysis and Modeling 75
4.2.4 Decision and Search Algorithm 80
4.3 Summary 83

5 Experimental Setup and Evaluation 85
5.1 Experimental Setup 85
5.1.1 Infrastructure 85
5.1.2 Cassandra Deployment 86
5.2 Experiments 87
5.2.1 Experiment Design 87
5.2.2 Database 88
5.2.3 Training Workloads 89
5.3 Experimental Results 89
5.3.1 Training Data and Model Generation 89
5.3.2 The Search Algorithm Alternatives 93
5.3.3 Use Case Example 95
5.4 Summary 96

6 Conclusion and Future Work 99
6.1 Summary of the Dissertation 99
6.2 Future Work 101

A Appendix 105

Bibliography 111

List of Figures

2.1 The Cloud Deployment Architecture, adapted from [ZCB10] 6
2.2 Relational DBMSs in the Cloud 9
2.3 Modular Architecture of Cloud Storage Systems 10
2.4 Historical Overview of the Family Diagram of Cloud Storage Systems . 12
2.5 Map/Reduce Computation Work-flow 16
2.6 CAP Theorem and Cloud Storage Systems 18
2.7 Tuning as a Continuous Process, adapted from [Sch12] 19
2.8 Database Tuning Reference Architecture 20
2.9 Self-tuning Cycle MAPE, adapted from [The05] 23
2.10 Examples of Analytical Models, adapted from [OK12] 27
2.11 Empirical Modeling Approach, adapted from [SSA08] 28

3.1 (Self-)Tuning Reference Architecture for Cloud Storage Systems 38
3.2 Illustration of the Complexity of Self-tuning Multi-layered Cloud Storage Systems 40
3.3 Divide and Tune 43
3.4 Self-Tuning Framework for Cloud Data Management Systems 44
3.5 Training Component Architecture 47
3.6 The XML schema for the Training Component Settings 49
3.7 The Performance of Underloaded Cassandra Cluster 50
3.8 Offline Mode of the Framework 56
3.9 Online Mode of the Framework 57
3.10 Offline-Online Mode of the Framework 59

4.1 Cassandra Ring, adapted from [Nee15] 65
4.2 Cassandra Column Family 65
4.3 Core Classes for the Training Component's Storage System Deployment 67
4.4 Sequence Diagram of the TrainingComponentManager's Main Loop . . 72
4.5 Sequence Diagram of the stopCluster Method 74
4.6 Core Classes for the Training Component's Workload Generation . . . 76
4.7 Sequence Diagram of the startLoad Method 77

5.1 Infrastructure used for the Prototype Deployment 86
5.2 Training Data: Latency of the Cassandra Cluster with Different Workloads 90
5.3 Latency of the Cassandra Cluster in Relation to the Number of Nodes & Write/Read Ratio 91
5.4 Input Measurements vs. Regression Analysis Result 92
5.5 Measurements vs. Prediction of Different Regression Techniques 92
5.6 Runtime for the Decision Component using Brute-force 93
5.7 Genetic Algorithm Result 94
5.8 Optimal Allocation of Nodes for Three Workloads 95
5.9 Framework Suggestion vs. Optimal Latency 96

List of Tables

2.1 Comparison of Related Work on Benchmarking Cloud Storage Systems 34

3.1 Training Data as Input for the Modeling Process 52

4.1 Comparison of Algorithms Considered for the Decision Component . . 81

List of Code Listings

4.1 Excerpt of the Cassandra Configuration File 69
4.2 Minimal Java Code for Creating Cassandra Key-space and Column Family 71
4.3 Excerpt of the Training Component Settings: Defining Database 71
4.4 Excerpt of the Training Component Settings: Defining Workload 75
4.5 Minimal Java Code for Write Operation 78
4.6 Excerpt Java Code for Using R with Java in our Framework 79
A.1 Schema Definition of the Training Component Settings 106
A.2 Training Component Settings 108

1. Introduction

The Cloud technology has been a trend in the IT world for many years. The Cloud is defined as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction." [MG11]. In other words, Cloud service providers support companies and individuals with elastic IT resources and services, such as servers, storage, and applications in pay-as-you-go model.

As the Cloud grew in popularity, large number of applications and services were provided solely on the Internet. Internet companies and Cloud service providers such as Google, Amazon and Yahoo! faced the problem of serving a rapidly increasing number of users. The traditional relational Database Management Systemss (DBMSs) could not scale to such workloads, making them unsuitable for many Cloud applications [ADEA12]. This lead to a new breed of data management and data processing systems [GGL03, CDG+08], called the Not only SQL (NoSQL) systems.

These systems are significantly different from conventional databases by focusing on providing scalability and availability to meet requirements of Cloud applications while disregarding typical DBMS features, such as complex query interfaces, transactional consistency management and stringent data models. Examples of these systems are Google's Bigtable [CDG+08], Yahoo!'s PNUTS [CRS+08], Apache's Hadoop Distributed File System (HDFS) [SKRC10], and Amazon's Dynamo [DHJ+07], etc. As more companies outsourced their data to the Cloud, relational storage on Cloud virtualized environment was supported, such as Amazon's Relational Database Service (RDS) and Microsoft's Cloud SQL Server [BCD+11]

Compared to the (self-)tuning of conventional database systems, which is intensively researched by industry and academia, the (self-)tuning of the Cloud NoSQL systems is still at its beginnings. Throughout the thesis, we focus on the (self-)tuning for Cloud clusters of NoSQL database systems serving online workloads.

In the following sections, we summarize the goal of the thesis and its contributions in the area of Cloud storage systems (self-)tuning. Then we provide an outline of the following chapters.

1.1 Goal of the Thesis

Though Cloud storage systems were developed to be self-managing regarding many aspects, such as load balancing, dynamically adding and removing nodes, there are many knobs that are left to be tuned according to applications' requirements. Examples of these knobs include: cluster size, replication factor, consistency level, data partitioning, and data placement strategies, etc. Furthermore, the distributed environment and the multi-layered architecture makes self-tuning for Cloud storage systems an even more complicated task. Moreover, if we have one Cloud database cluster serving one or more applications, which are serving several workloads with different optimization goals, the question is how to tune the database cluster to best achieve the optimization goals of all workloads. To answer this question, we define the goal of the thesis as:

> *To design, implement and evaluate an approach for self-tuning Cloud storage cluster serving different workloads. The framework should be database system agnostic, platform independent, extendable, and should allow online refinement.*

To achieve this goal, we made the following contributions:

- As a precondition for the proposed framework, we relate tasks of (self-)tuning to layers and sub-clusters within typical Cloud storage architecture. We define a guideline for typical Cloud storage system (self-)tuning processes.
- We build a training component that automates the process of testing and monitoring a Cloud storage cluster with different configurations and different workloads. This component generates training data needed to model the performance of Cloud storage cluster and can be used as a benchmark.
- We build a cost estimation component to predict performance metrics based on the empirical data-driven approach. Our results on modeling the performance characterized by latency in relation to write/read ratio and cluster size shows that typical regression techniques provide good predictive models for this case.
- Based on measured and/or modeled performance of applications, we design a decision component and evaluate two search algorithms; brute-force and genetic algorithm. Our evaluation shows that though the brute-force provides the optimal solution, its cost grows proportionally with the search space. Our genetic algorithm based approach provides near optimal solution with a smaller cost compared to the brute force approach.

- We address the essential problem of tuning the size of (sub-)clusters of the targeted workload/workloads. Our framework supports creating an optimal setup of sub-clusters for all workloads, indicated by a global minimum of latency, and within resources restrictions.

1.2 Outline

In the following chapters, we provide details about the design, implementation, and evaluation of our framework. The chapters are structured as the following:

Chapter 2: This chapter introduces the foundations, which are necessary to understand the research problems and approaches investigated in the thesis. It includes details about Cloud data management, database self-tuning, and performance modeling. It also includes an overview of related work.

Chapter 3: In this chapter, we motivate the self-tuning problem in the context of Cloud data management. Then, we introduce an architecture of a self-tuning framework for Cloud storage systems clusters. This chapter includes discussions about design decisions of different components of the framework architecture.

Chapter 4: This chapter reveals implementation details for our framework. It includes details about the used technologies and systems. It also discusses the alternatives for implementing the framework components.

Chapter 5: In this chapter, we present the evaluation results of our framework. This chapter provides an outline of the conducted experiments and their goals. At the end, we provide a use case scenario of the framework and discuss the results.

Chapter 6: This chapter summarizes the thesis, outlines the contributions, and provides an insight into future research directions.

2. Background and Related Work

This chapter shares material with paper "Cloud Data Management: a Short Overview and Comparison of Current Approaches" [MBS12]

In this chapter, we provide the background concepts necessary to understand the domain of Self-tuning for Cloud storage systems. Furthermore, we present definitions and principles of the used techniques to achieve self-tuning in our framework.

First, we provide a literature review of Cloud data storage systems. Then, we provide basic concepts of (self-)tuning, and performance modeling for storage systems.

2.1 Cloud Data Management

In this section, we provide an overview of Cloud data management. We start by presenting an overview of the Cloud computing environment and its application needs. Then, we provide an overview of the current state of the art for Cloud storage systems.

2.1.1 Cloud Environment

The Cloud technology has been a trend in the IT world for many years. Cloud providers aim to free companies from worrying about their IT resources just like the electrical grid freed companies from generating their own electricity[FZRL08]. The Cloud is defined as a shared pool of computing resources that can be provisioned and released with minimal effort or service provider interaction [MG11]. This pool of computing resources is offered in a pay-as-you-go network access. This means for both companies and individuals, the ability to get resources whether storage space, computational power, or Software over the Internet.

The Cloud model is deployed in several layers. The first one is ***Infrastructure as a Service*** layer. This layer is responsible for managing physical resources and creating a pool of storage and computing power by means of virtualization techniques. The second layer is the ***Platform as a Service***, which consists of operating systems and application's platforms. This layer minimizes the burden of deployment onto the virtual machines containers of the underlying layer. The final layer is the ***Software as a Service*** layer, which consists of actual Cloud applications. Different from traditional applications, they can leverage the Cloud features to achieve better performance, availability and lower operating costs [ZCB10]. The Cloud deployment architecture is illustrated in Figure 2.1.

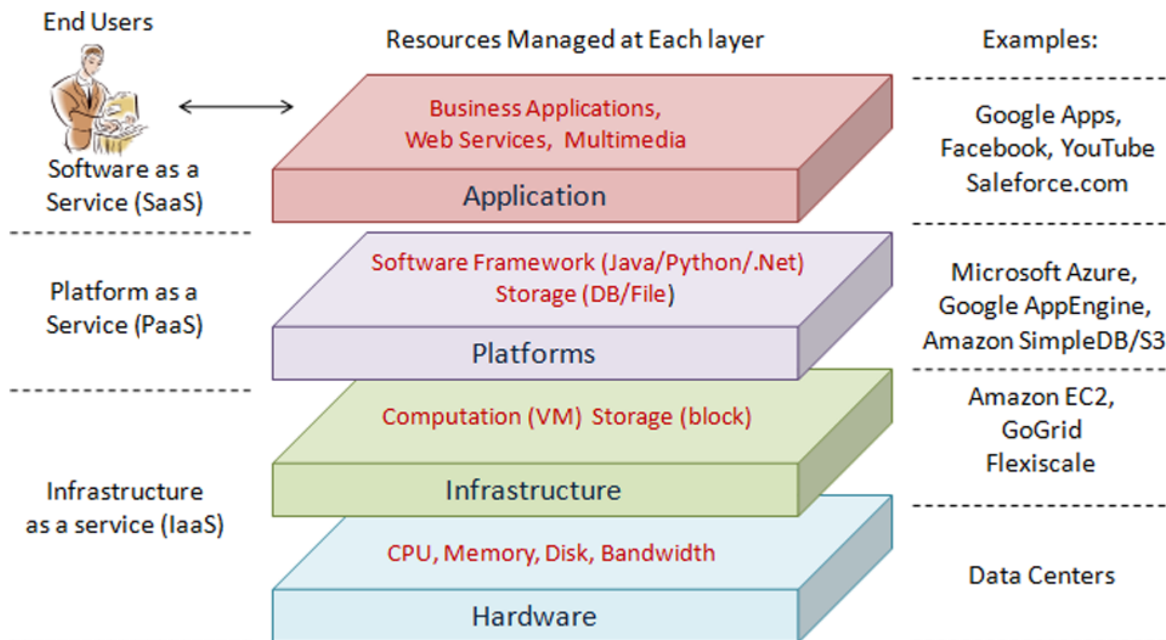


Figure 2.1: The Cloud Deployment Architecture, adapted from [ZCB10]

Cloud computing has several characteristics summarized in the following points [SLMBA11, ZCB10]:

Multi-tenancy and Shared-resources Pooling: Multi tenancy means sharing resources among several tenants i.e. applications or users. Multi tenancy in the Cloud is provided on different levels of abstraction and isolation models. An example of multi tenancy is a physical server hosting several virtual machines belonging to different users.

On-demand Broad Network Access: A Cloud customer can provision resources without requiring human interaction with each service provider. The access and provisioning of resources is available through network access by means of standard mechanisms available through the heterogeneous thin and thick client platforms starting by mobile phones and tablets to laptops and mainframes.

***Dynamic Resource Provisioning*:** Resources can be elastically assigned and released with minimal effect on performance or availability of the services using them during the assignment/release process.

***Geo Distribution*:** Cloud providers typically use a distributed network of resources over several data centers and possibly in different continents. A service provider can leverage geographical distribution to achieve maximum service utility.

2.1.2 Cloud Requirements for Storage System

As the Cloud grew in popularity, large number of applications and services were provided solely on the Internet. Internet companies and Cloud service providers such as Google, Amazon and Yahoo! faced the problem of serving a rapidly increasing number of users. For the storage purposes, traditional relational DBMSs could not scale to such workloads, making them unsuitable for hosting many Cloud applications [ADEA12]. Google pioneered Cloud storage systems by creating Google File System [GGL03] and Bigtable [CDG+08] as the storage back-end for over 60 services and products, such as Google Finance, Google earth, and Google Analytics, etc. Design decisions regarding architecture, functional and non-functional requirements were made based on analyzing the environment in which the storage system works and the targeted applications' workloads that it serves. More storage systems were designed in the same manner by companies, such as Yahoo!, Amazon, Facebook and Apache.

Next we provide an overview of the application requirements of Cloud storage systems. We derive these requirements by reviewing different Cloud storage systems [CRS+08, DHJ+07, CDG+08, SKRC10, The15c, Hel07, SLMBA11]. Each of these systems have special application requirements, however, they share the following:

***Scalability and Elasticity*:** The workload of the targeted applications and services can witness seasonal, unpredictable, or planned heavy processing. In all cases, the storage system must be able to scale with minimal operational effort and minimal impact on its performance or the performance of any service that is using it. It also should support scaling down, in a similar manner.

***Fault Tolerance and High Availability*:** The storage system will run on a cluster of hundreds or even thousands of inexpensive hardware components. Having a failure in one or many components is the normal case that the storage system will face rather than the exception. Therefore, failure detection, and recovery should be incorporated in the architectural design of the Cloud storage system. Operations should continue with minimal or no interruption.

***Response Time and Geographic Scope*:** The storage system will be used internally by other services of the Cloud provider, which are requested by users in a wide range of geographical space. It should meet the Service Level Agreement (SLA) of response time and respond to all users with low latency.

***Data Volume and Scalability*:** The storage system, whether it is a file system or a structured storage system, will deal with large sizes of data that is constantly growing. Single file size can grow to multi megabytes of data; for this the design parameters down to the level of Input/Output (I/O) operations and block sizes must be adapted.

***Load and Tenant Balancing*:** This requirement is essential to achieve effective resource utilization and cost optimization for a storage system that is working on a network of distributed computing nodes.

***Evolving Schema*:** The data model of the storage system should provide applications with flexibility and freedom to evolve how information is structured and should allow and support schema evolution.

***Security and Privacy*:** Outsourcing data to the Cloud raises concerns for companies and individuals regarding data security and privacy. Cloud storage systems should provide secure storage that allows storing and querying encrypted data when needed. The Cloud also raises concerns regarding data compliance, which specifies data ownership and protection. Data compliance becomes complicated when the data of an organization is stored in a different country that applies different privacy laws; such as an EU company storing its data in the US.

As the Cloud grew in popularity, more companies started outsourcing their data storage and processing. However, the needs of some of these companies regarding transaction support and data consistency required the use of the more mature and ACID-compliant relational storage systems. The Cloud storage landscape evolved to include relational DBMSs as service. A number of research efforts were directed toward breeding both approaches into a hybrid system such as HadoopDB [ABPA+09].

In the following subsections, we provide an overview of both newly designed and traditional DBMSs in the Cloud.

2.1.3 Relational Database Management Systems as a Service

In this approach, each customer gets an own instance or instances of a DBMS that runs on virtual machines of the Cloud service provider. Systems in this approach support full Atomicity, Consistency, Isolation, Durability (ACID) requirements with the disadvantage of limited scalability. This means, if the application requires more resources than the provided classes of instances, the customer must implement partitioning on the application level, then use different instances for each partition. An example of such systems are the Amazon RDS which provides MySQL, Oracle, and PostgreSQL.

Compared to relational DBMSs deployed on premise, relational DBMSs as a service have several advantages and drawbacks. On the one hand, the Cloud supports pay-as-you-go business models, which means low up-front cost and on demand scalability. The Cloud provides relational DBMS as a managed service where many of the database

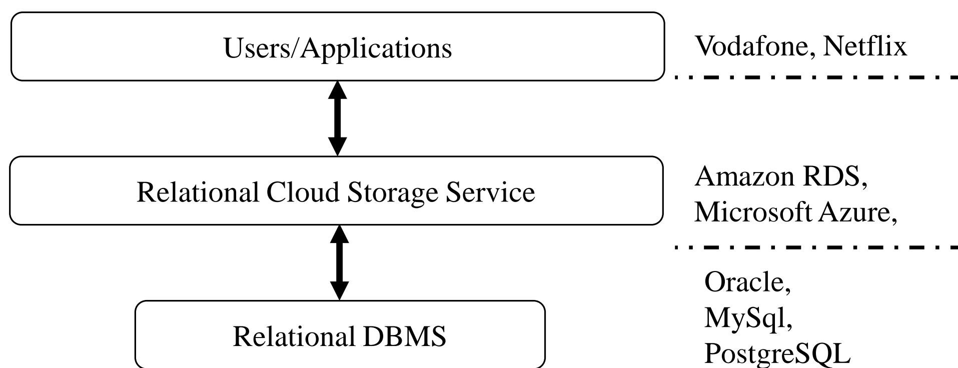


Figure 2.2: Relational DBMSs in the Cloud

administration tasks, such as patch management and backups are responsibility of the service provider, which leads to lower staff cost. From the technical side, relational DBMSs as a service support different availability zones and read only replicas to increase availability. On the other hand, issues, such as data security and protection, service outages and data loss raise big concerns for companies considering outsourcing their data to the Cloud.

Compared to their NoSQL counterparts, relational DBMSs have the advantage of familiar tools and programming models, which is important for companies with legacy architecture. Besides these features, well-established relational DBMSs qualities such as query optimization, built-in tuning, and transaction support are essential for many applications. However, current relational DBMSs are considered not Cloud-friendly because of their limitation in scalability support [ADEA12]. When it comes to handling the growing scale of data and the number of requests, current relational DBMSs fail to provide adequate tools and guidance.

2.1.4 Not only SQL Storage Systems

Properties of the Cloud environment such as multi-tenancy and components failure, etc. and the needs of its application such as scalability, availability, and fault tolerance, etc. resulted in a new breed of data storage systems. These systems were primarily developed for internal use by companies such as Google (Bigtable) , Amazon (Dynamo), Facebook (Cassandra), etc. In the following subsections, we provide a detailed overview of the Cloud NoSQL storage systems.

2.1.4.1 Modular Layered Architecture

For the implementation of traditional DBMSs, developers and researchers suggested multiple-layered architectures. Among the most referenced to, in literature, are the ANSI SPARC architecture, and the five level architecture. More details on each can be found in [SSH08, Här87]. Such a reference architecture for Cloud data management systems does not exists. As a result from a survey [MBS12] that we conducted on Cloud storage systems, we present the architecture in Figure 2.3.

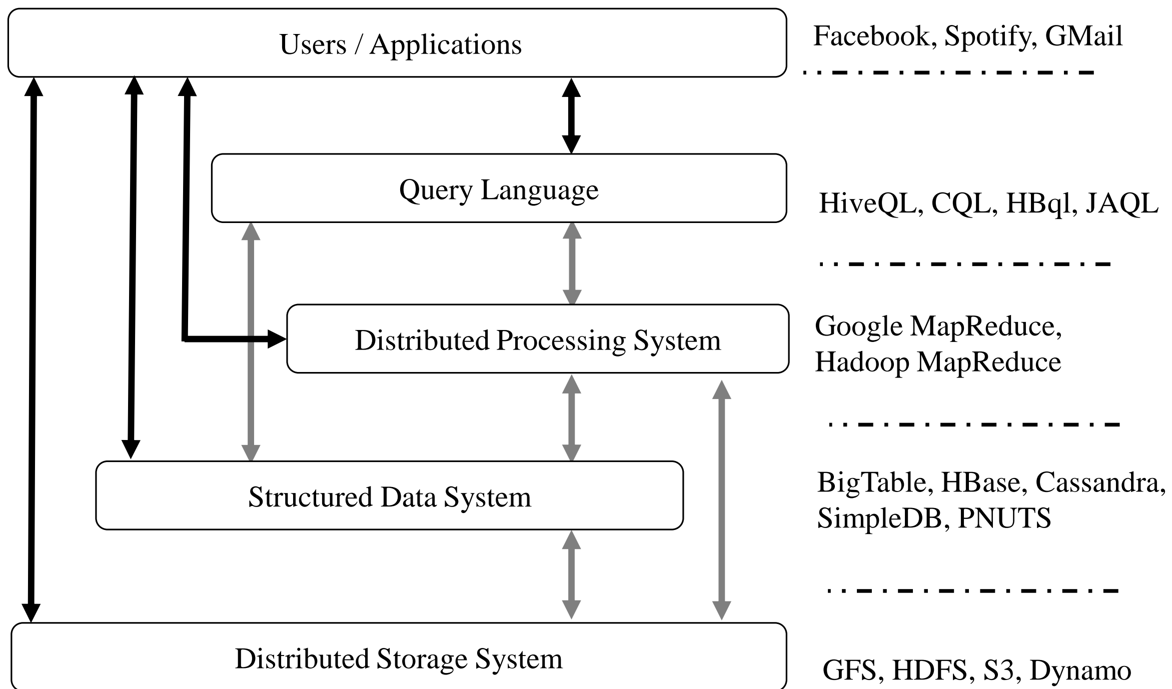


Figure 2.3: Modular Architecture of Cloud Storage Systems

We group NoSQL storage systems and services in a modular architecture made of four layers on top of which lays the application layer. In deployment, a combination of these interconnected systems and services can be formed according to application needs.

1. Distributed File System:

   The essential layer of this architecture is a distributed file system. Systems of this layer were developed for internal use by cloud providers and are typically not delivered as a public service. It is responsible for providing availability, scalability, fault tolerance, and high performance data access. We divide systems of this layer into three categories:

   - Distributed file system, such as Google File System (GFS) [GGL03] and HDFS [SKRC10], etc.
   - Cloud-based file services, such as Amazon's Simple Storage Service (S3) [The15a].
   - Peer to peer file system, such as Amazon's Dynamo [DHJ+07].

2. Structured Data System:

   The second layer consists of structured data systems. Systems of this layer provide simple data models, such as key-value pairs and support various Application Program Interfaces (APIs) for data access and different protocols, such as Simple Object Access Protocol (SOAP) and Hypertext Transfer Protocol (HTTP). Examples of systems of this layer include Google's Bigtable [CDG+08], Cassandra [LM10], and SimpleDB [The15b].

3. Distributed Processing System:

   The third layer includes distributed processing systems, which are responsible for more complex data processing, e.g., analytical processing, mass data transformation and DBMS-style operations, such as joins and aggregations. Google pioneered systems developed for this layer by introducing their Map/Reduce Framework [DG08], which became the dominant processing paradigm for such operations in Cloud and Big data management.

4. Query Language:

   The fourth layer includes the language support. Since the underlying systems do not support the relational model, Structured Query Language (SQL) is not supported. However, developers of these languages tried to mimic SQL syntax for simplicity providing languages, such as Cassandra Query Language (CQL) [The15d]. More recently developed systems or languages, such as Hive Query Language (HQL) [TSJ+09], support joins and aggregations. However, these operations are internally translated into Map/Reduce operations [TSJ+09, LLH+11].

Compared to their relational counterparts, many NoSQL systems are not off-the-shelf systems. Their deployment requires extensive knowledge of internals, tuning knobs and performance trade-offs. Moreover, these systems complement each other and work together to provide different sets of functionalities. In the following section, we provide an overview of the connections between the systems of the different layers and their possible combinations.

2.1.4.2 The Cloud Data Management Systems Family Diagram

The purpose of this subsection is to provide an overview of how NoSQL developed and the dependencies between the different component systems and services. The development of NoSQL systems was heavily driven by industry. Google pioneered this development by introducing their distributed file system [GGL03] and Bigtable [CDG+08]. Later on, developers at Google provided the Map/Reduce Framework [DG08] for parallel processing over big data. In Figure 2.4, we visualize historical development of Cloud storage systems and services until 2010. The aim of the figure is not to provide an extensive view of all Cloud storage systems, rather provide an overall view and depict the connections among the different systems. We use a dotted arrow to illustrate that a system uses one or more concepts from another system, such as data model or processing paradigm. Example of this is Cassandra using the Bigtable's data model. A solid arrow refers to one system using another one, such as Hive using HDFS.

This family diagram starts at the left side with distributed file systems. Then we have structured storage systems with API. Then comes systems that support simple query language, such as SimpleDB. Next, we have the structured storage systems with support of Map/Reduce and query language, such as Cassandra and HBase. Finally, we have systems with sophisticated query language and Map/Reduce support, such as

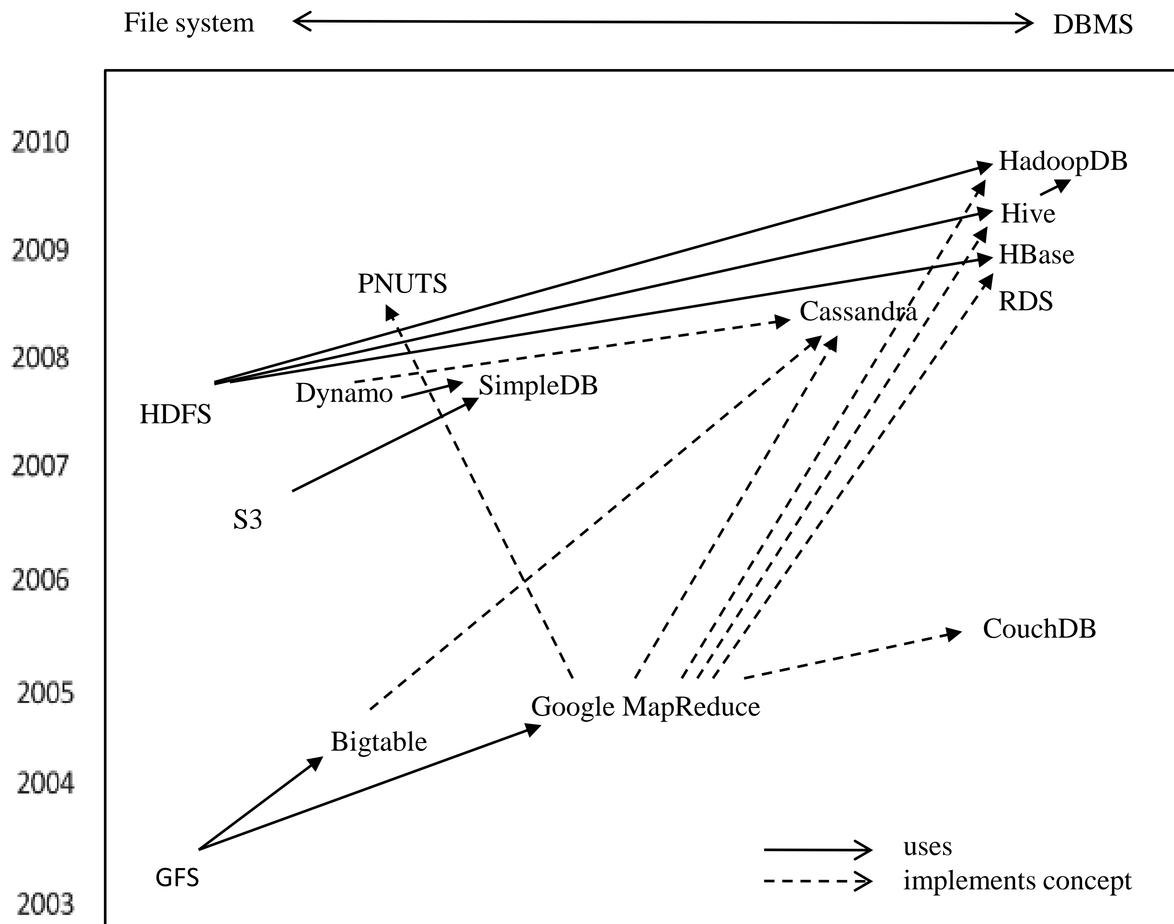


Figure 2.4: Historical Overview of the Family Diagram of Cloud Storage Systems

Hive [TSJ+09], and HadoopDB [ABPA+09]. Along this spectrum of Cloud storage that starts with file system functionalities and ends with fully-fledged data management, the different storage systems share the following features:

- Run on a cluster of commodity hardware where node failure is the normal.
- Data is partitioned to achieve scalability.
- Data is replicated to achieve availability.
- Non relational key-value store is the dominant data model.
- CAP theorem tradeoffs leads to a spectrum of consistency models.

These features affect performance and govern tuning trade-offs. In the following sections, we provide an overview of each of them.

2.1.4.3 Cluster Management

Cloud storage systems are expected to run on a cluster of commodity hardware where component failure is the normal rather than the exception. As a basic functionality, cluster management should support failure detection and recovery. In alignment with the Cloud elasticity, it should also support adding and removing nodes with minimal effort. Another important function for cluster management is load balancing which allows resource utilization. Though the different Cloud storage systems use different strategies to achieve the previous functions, their cluster management approaches fall under the two basic approaches:

***Master-slave Approach*:** In master-slave configuration, a single master maintains system's meta-data and controls system-wide activities. This approach is used in Bigtable, HBase.

***Peer to Peer Approach*:** In peer to peer configuration, cluster nodes are architecturally identical. This approach is used in Dynamo, Cassandra.

2.1.4.4 Partitioning

Partitioning or data sharding is used by Cloud storage systems to achieve scalability. There is a variety of partitioning schemes on different levels. Some systems partition data on the file level while others horizontally partition data on the key space or table level. Examples of systems that partition data on the file level are systems of the distributed file system layer, which partition each file into fixed sized chunks of data. The second class of systems, which partition key space or tables, use schemes, such as list, range, and hash partitioning [SFKS10, Cyr02]: Partitioning or data sharding is used by Cloud storage systems to achieve scalability. There is a variety of partitioning schemes on different levels. Some systems partition data on the file level while others horizontally

partition data on the key space or table level. Examples of systems that partition data on the file level are systems of the distributed file system layer, which partition each file into fixed sized chunks of data. The second class of systems, which partition key space or tables use schemes, such as list, range, and hash partitioning [SFKS10, Cyr02]:

***List Partitioning*:** A partition is assigned a list of discrete values. If the key of the inserted tuple has one of these values, the specified partition is selected. Example of a Cloud data management system using list as the partitioning scheme is Hive.

***Range Partitioning*:** The range of values belonging to one key is divided into intervals. Each partition is assigned one interval. A partition is selected if the key value of the inserted tuple is inside a certain range. Example of a system using range partitioning is HBase.

***Hash Partitioning*:** A hash function is applied on key values to determine their partition. The output of the hash function is assigned to different partitions. Example of a system using hash as the partitioning scheme is PNUTS [CRS+08].

Some of the Cloud storage systems use a composite partitioning technique. An example of such systems is Dynamo [DHJ+07], which uses a composite approach of hash and list partitioning. Some systems support partitioning data several times using a different partitioning scheme each time. Example of such systems is Hive [TSJ+09], which allows two levels of partitioning. After horizontally partitioning the table based on column values, each partition can be hash partitioned into buckets for storage in the underlying HDFS.

Choosing an order preserving partitioning scheme has an impact on performance. Examples of systems using order preserving techniques are Bigtable and Cassandra. An important issue regarding partitioning is the assignment of data partitions to nodes. Most systems use random assignment. However, depending on the data, this can raise bottleneck problems. Several approaches are used by different systems to overcome this problem. Dynamo focuses on achieving uniform distribution of keys among the storing nodes [DHJ+07]. Cassandra provides a load balancer that analyzes load information to lighten the burden on heavily loaded nodes [LM10].

2.1.4.5 Replication

Replication is used by Cloud storage systems to improve availability. Replicas of data are stored on more than one node and probably more than one data center depending on the application needs and the used replica placement strategies. Cloud storage systems use the following replica placement strategies [Hew10]:

***Rack Unaware Strategy*:** Also known as the Simple Strategy. It places replicas within one data center without configuring replica placement on certain racks.

***Rack Aware Strategy*:** Also known as the Old Network Topology Strategy. It places replicas on different racks within one data center.

***Data Center Aware Strategy*:** Also known as the New Network Topology Strategy. It places data across different data centers, allowing clients to specify in their applications how replicas are placed cross the different data centers.

The replication factor, that determines how many replicas are made of data items, is handled in different ways. Some systems do not reveal it to users, e.g., Amazon S3 and SimpleDB. Most systems allow users to set the replication factor. One example is HDFS [SKRC10], which allows clients to set the replication factor on the file level.

Replication improves system robustness against node failures. When a node fails, the system (transparently to the user) reads data from other replicas. Another gain of replication is increasing read performance with the help of a load balancer that directs requests to a data center close to the user. Replication has a down side when it comes to updating data. The system has to update all replicas. This leads to very important design considerations that impact the availability and consistency of data. The first one is whether to make replicas available during updates or to lock them until data is consistent across all of them. Most systems in the Cloud choose availability over consistency. We will discuss this trade-off and related consistency models, in more details, in Section 2.1.4.8. The second design consideration is when to perform replica conflicts resolution i.e. during writes or reads. If conflict resolution is done during write operations, writes could be rejected if the system can not reach all replicas or a specified number of them within a specific time. Example of that is the WRITE ALL operation in Cassandra [Nee15], where the write fails if the system could not reach all replicas of data. However, some systems in the Cloud choose to be always writeable and push conflict resolution to read operations. An Example of that is Dynamo [DHJ+07], which is used by many Amazon services, such as the shopping cart service where customer updates should not be rejected.

2.1.4.6 Data Models and Data Access

Just as different requirements of the Cloud applications and its environment led to the previously described architectural and implementation decisions, they also led to different data models. The main data models used by Cloud systems are divided into two categories. The first one is the relational data model. The second category is the key-value stores, under which comes the following sub-categories:

***Row Oriented*:** Data is organized as containers of rows that represent objects with different attributes. Access control lists are applied on the object or container (set of rows) levels.

***Document Oriented*:** Data is organized as collection of self described JSON documents. Documents are the unit of access control. Example of systems using Document oriented data model as a data model is CouchDB [CLS10].

***Wide Column*:** In this model, attributes are grouped together to form a column family. The schema for this model is typically extensible even after creation and insertion of data. Example of systems supporting such model are Cassandra and HBase.

Data querying capabilities in Cloud storage systems vary in accordance with the different data models. Some systems provide querying option on a single data object, such as in Amazon's S3. Other systems provide querying capabilities on a single container of objects (or table of records), such as SimpleDB, HBase and PNUTS. At the end of the Cloud storage systems' spectrum, sophisticated operations, such as joins and aggregations are supported. These operations are internally transformed and executed by parallel processing paradigms such as Map/Reduce. More details about the parallel processing paradigms for Cloud storage are provided in the coming section.

2.1.4.7 Parallel Data Processing in the Cloud

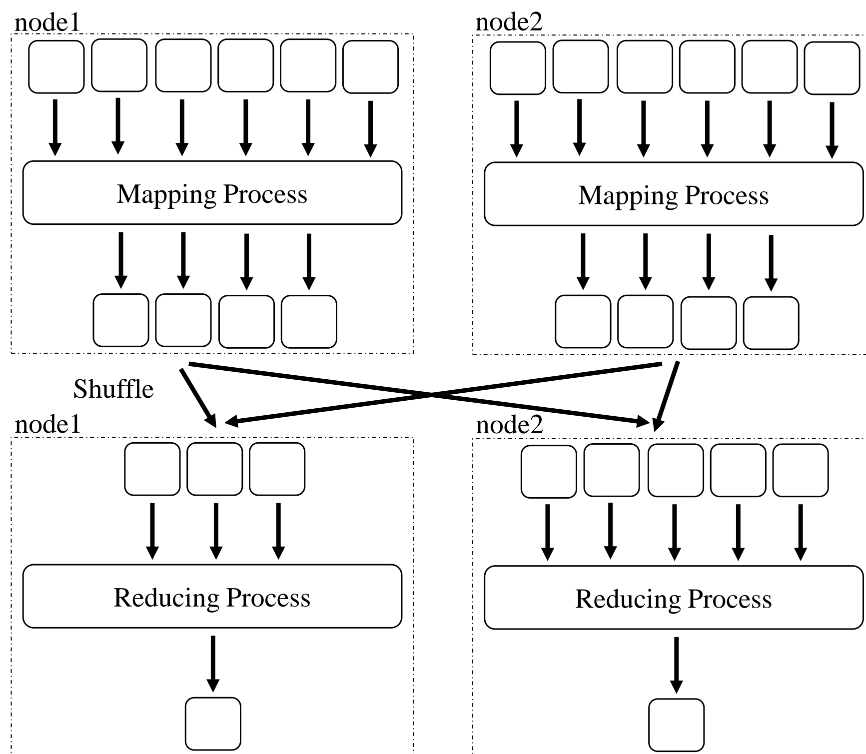


Figure 2.5: Map/Reduce Computation Work-flow

Large volumes of data are being produced in different fields from scientific experiments, and sensor data to Internet and mobile phone-data. Before the Cloud era, several DBMSs were implemented to support geographically-distributed data storage and large-scale processing (the parallel and distributed relational DBMSs). However, the largest part of data growth is with data in unstructured form that is why new processing paradigms with flexible data models were needed [FE10, GR13].

A similar paradigm to the Cloud, grid computing, is primarily used in scientific environment for big data processing [FE10]. In the Cloud, the Map/Reduce framework represents the state of the art for big data processing. As already mentioned, the Google Map/Reduce [DG08] pioneered Cloud parallel processing frameworks. It gained popularity, and was open sourced by Hadoop in 2006.

Map/Reduce is a programming paradigm that is used for distributed computations on data sets of key value pairs. The Map/Reduce framework maximizes the utilization of the Cloud environment computing nodes by enabling parallel execution of user defined Map/Reduce jobs. A single Map/Reduce job is executed in two stages. The intermediate output from the first stage is shuffled and assigned to reducers. The work-flow of Map/Reduce jobs is illustrated in Figure 2.5.

The Map/Reduce framework is independent of the the underlying storage layers and can work with different layers. It is also fault tolerant, which means that the process of Map/Reduce does not fail in the case of Map/Reduce jobs failure. The re-execution of the failed tasks takes place automatically and quickly [CDG+08].

2.1.4.8 CAP Theorem and Consistency Levels

Tightly related to key features of Cloud data management are discussions on the Consistency, Availability, Partition Tolerance (CAP) theorem [FGC+97]. The CAP theorem states that consistency, availability, and partition tolerance are systematic requirements for designing and deploying distributed systems. If the system needs to be scalable, one of the three properties must be sacrificed. In the context of Cloud data management, the CAP theorem leads to design decisions regarding the consistency and availability resulting in relaxed and flexible consistency models. Based on this, the following consistency models, as illustrated with examples in Figure 2.6, are provided by Cloud storage systems:

***Atomicity, Consistency, Isolation, Durability (ACID)*:** With ACID, users have consistent overview of data before and after transactions. A transaction is Atomic, i.e., when one part fails, the whole transaction fails and the state of data is unchanged. Once a transaction is committed, it is protected against crashes and errors. Data is locked while being modified by a transaction. When another transaction tries to access locked data, it has to wait until data is unlocked. Systems that support ACID are used by applications that require strong consistency and can tolerate losing availability. However, providers of Cloud data management systems (e.g. Amazons RDS) support options for enhancing availability, such as read replicas and multi-zones availability.

***Basically Available Soft state Eventual consistency (BASE)*:** The storage system does not guarantee that all users see the same version of data item but guarantees that all of the users get response from the system even if it means getting a stale version. BASE is used by applications, which can tolerate weak consistency to have higher availability. Examples of such systems are SimpleDB and CouchDB.

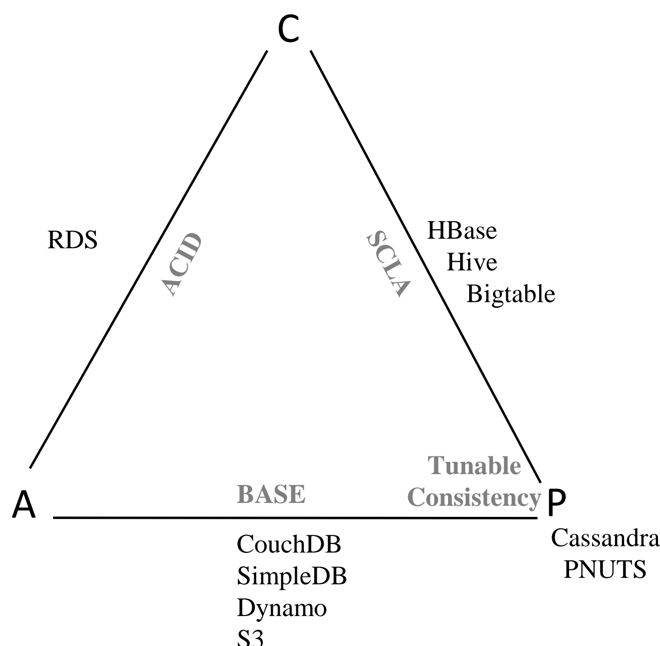


Figure 2.6: CAP Theorem and Cloud Storage Systems

***Strongly Consistent Loosely Available (SCLA)*:** The storage system, in this case, provides better availability than ACID and stronger consistency than BASE. It is used by applications that choose higher consistency to sacrifice availability to an extent. Examples of systems supporting SCLA are HBase [The15f] and Bigtable [CDG+08]

Tunable Consistency: The storage system allows configuring the consistency level. For each read or write request, the consistency is configured by setting the number of replicas or storage-nodes the request touches. This allows the system to work in high consistency or high availability and other degrees in between. An example of a system supporting tunable consistency is Cassandra [LM10].

2.2 Database Tuning and Self-tuning

Tuning and self-tuning are parts of every-day processes in IT frameworks. The tuning and self-tuning for database systems are a well established research field which has been heavily studied by both academia and industry. In the coming subsections, we provide an overview of system tuning in general and database tuning in more details.

2.2.1 The General Process of System Tuning

System tuning starts by identifying problems in the performance of the systems. Unfulfilled thresholds regarding response time, latency, or consumed computing resources are typical problems for computing systems that could initiate the tuning process.

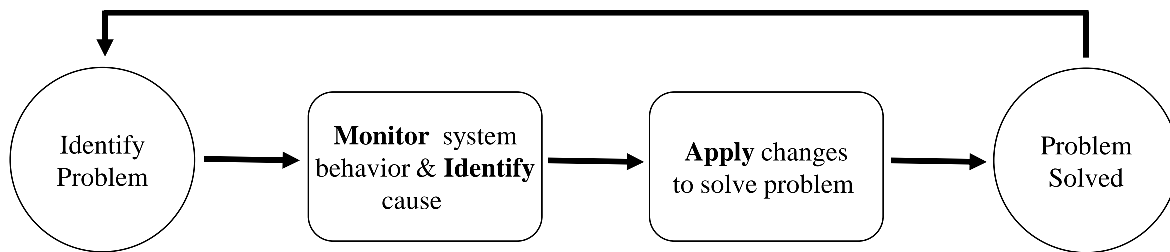


Figure 2.7: Tuning as a Continuous Process, adapted from [Sch12]

The next step in the tuning process is observing the system performance and measuring relevant quantities e.g., time spent in memory or hard disk look-ups, I/O, and network communication. This also includes identifying the part of the system causing the problem and assessing different possibilities for a solution.

The final step includes applying changes, such as adjusting system parameters, adding resources, etc.

The different steps of a system tuning process are depicted in Figure 2.7. In the next section, we provide an overview of the tuning process in the context of database systems.

2.2.2 Database Tuning

According to [SB02], *database tuning* is "the process of making database application run more quickly. *More quickly* usually means higher throughput though it may mean lower response time for some applications." Database tuning evolved to include resource consumption, e.g., Central Processing Unit (CPU), memory, disk, and energy, etc. The database tuning problem is driven by observed or expected workloads and optimization goals and results in tackling knobs in different parts of the IT infrastructure. In the following, we describe the basic principles for tuning database systems and the reference architecture for the tuning process.

2.2.2.1 Database Tuning Principles

The following tuning principles are guidelines for the overall database tuning process as suggested by [SB02]:

- Think globally, fix locally.
- Partitioning breaks bottlenecks.
- Start-up costs are high, running costs are low.
- Resource-aware task assignment between the DBMS and applications.
- Be prepared for trade-offs.

Besides these general principles, the literature for database tuning includes rule of thumbs and tuning guides specific to certain DBMS or application scenario.

2.2.2.2 The Reference Architecture for Database Tuning

Database tuning requires working with different components of a database management systems. In addition to that, it requires handling both the application layer and the underlying operating systems and hardware layers. Based on the five level database architecture [SSH08, Här87], the reference architecture of database systems tuning is illustrated in Figure 2.8. Examples of possible tuning knobs in each layer are given in the right side of the figure.

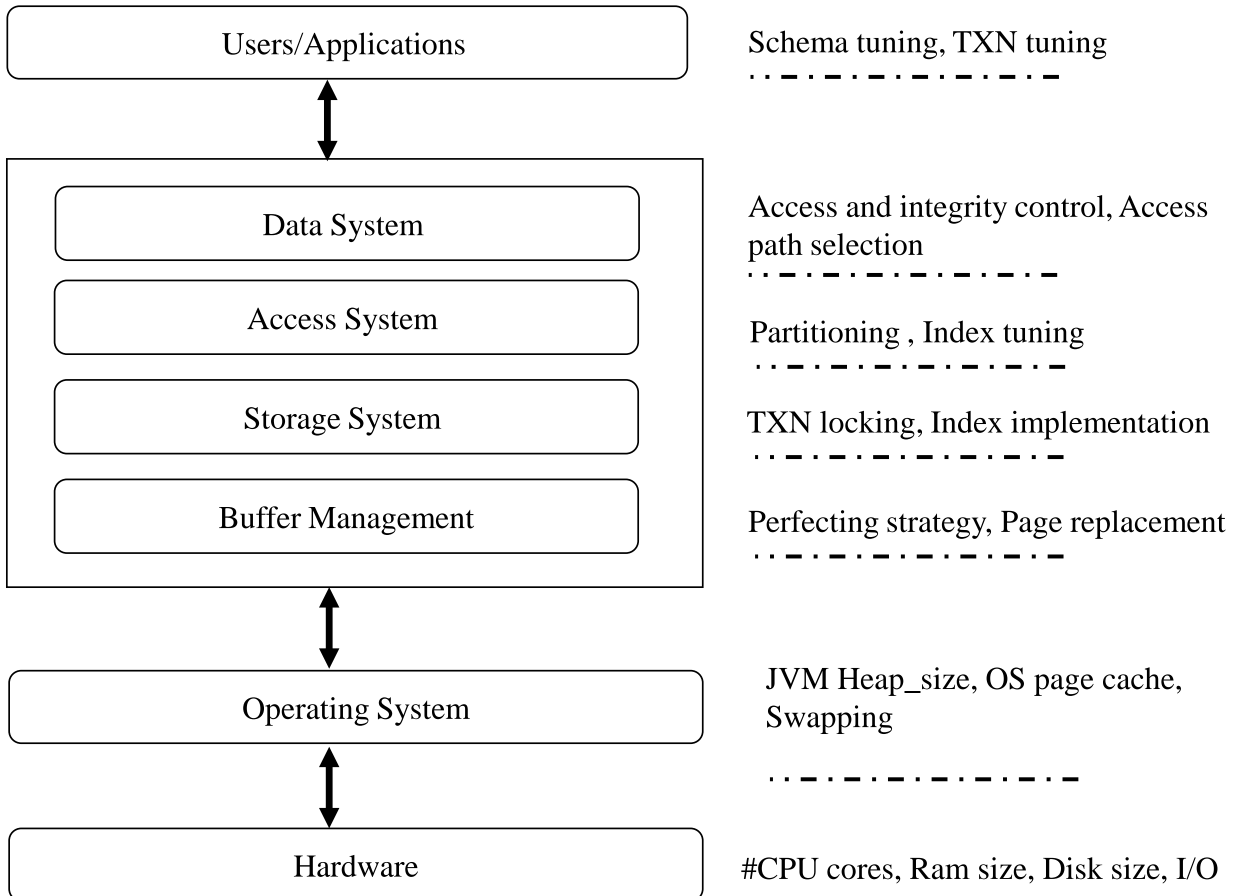


Figure 2.8: Database Tuning Reference Architecture

Efficient database tuning involves a good knowledge about the ***application layer*** to identify how the database is used and what the optimization goals/performance thresholds are. Possible tuning knobs in this layer includes:

- Transaction tuning: e.g., changing isolation levels, partitioning transaction.
- Query tuning: e.g., rewriting semantically equivalent queries.
- Schema tuning: e.g., de-normalization, partitioning.

It also involves a good knowledge about ***architecture and features of the used DBMS*** to identify the tuning possibilities. Traditional DBMS components tuning includes the following:

- Data system tuning: e.g., query translation, optimization, access path selection.
- Access system tuning: e.g., index tuning, table partitioning.
- Storage system tuning: e.g., page and file size tuning, deadlock handling.
- Buffer management tuning: e.g., page pre-fetching strategies, memory usage.

Database tuning process continues in the ***operating system layer*** and involves operations, such as threads tuning and driver configuration. The last part, involved in this process, is the ***hardware layer***. Though this could be the first layer that companies would work on by applying the *kill-it-with-iron* principle. This principle improves the database performance only in a linear factor and contradicts possible energy saving and green initiatives goals. Tuning possibilities include changing hardware components, configuring disk arrays, processor utilization, etc.

2.2.3 Database Self-tuning

Automating the database tuning process means delegating the tuning tasks to the database system itself or to a software layer that connects to the applications and the DBMS. The self-tuning shift for database applications was motivated by several reasons [CW06]:

- The increasing complexity of multi-tenant monolithic applications and services.
- The increasing complexity of DBMSs administration and tuning which provides hundreds of tuning knobs.
- The increasing Total Cost of Ownership (TCO) for DBMS-based IT solution with the cost for hiring experts in system tuning and management being the dominant.

The self-tuning principles, which we use for building our framework follow the rules defined in the coming sections. First, we provide an overview of the self-tuning principles, then the IBM initiative on autonomic computing process and the self-tuning paradigms. After that, we provide an overview of (self-)tuning for Cloud storage systems.

2.2.3.1 Self-Tuning Principles

Based on [CW06, CW09a, Sch12], the self-tuning principles are the following points.

***Static vs. Online Self-tuning*:** The difference between the two approaches is in three areas. The first one is regarding the frequency of the tuning process. The static tuning is performed once and initiated by the system administrator. The online self-tuning is performed continuously. The second difference is regarding the architecture. Where the static self-tuning is decoupled from the DBMS to a large degree and is more likely to be provided by a separate tool (framework or Software layer), the online self-tuning is more coupled to the DBMS. The third difference is regarding the scope of the tuning process. Where the static tuning covers slowly changing or stable properties of the database system, such as physical configuration, the online self-tuning covers more frequently changed parameters, such as memory buffer allocation.

***Self-Tuning Feedback Loop*:** This loop is based on the continuous process principle of system tuning; described in Section 2.2.1. IBM published a blueprint in an attempt to standardize the different step of this loop. In the next subsection, we will provide detailed description of this cycle.

***Trade-off Elimination*:** The self-tuning system should be able to cope with trade-offs that arise depending on the workload or application scenario. Trade-off elimination is transformed into parameter tuning. In Section 2.2.4.2, we provide an overview of Cloud storage systems trade-offs.

***Self-tuning Overhead*:** Additional processing power is required to make the tuning decision and additional space is required to store information needed and resulted from the self-tuning process. The two additional costs cause an overhead for the system. This overhead must be smaller compared to the benefit from the self-tuning process.

2.2.3.2 Self-tuning MAPE Cycle

IBM provided a blueprint for the self-tuning process known as Monitor, Analyze, Plan, Execute (MAPE) cycle. This blueprint defines the tasks involved in a self-tuning framework and not its architectural components and can have different implementations in different IT-Infrastructures [The05]. The MAPE cycle as illustrated in Figure 2.9 includes the following parts that share a common knowledge-base:

***The Monitor Function*:** This function provides mechanisms to collect information from the managed resources. Managed resources can be software or hardware components. Collected information include details about resources configuration, status, throughput, etc. The monitor function performs further operation on the collected information, such as filtering and aggregation, etc.

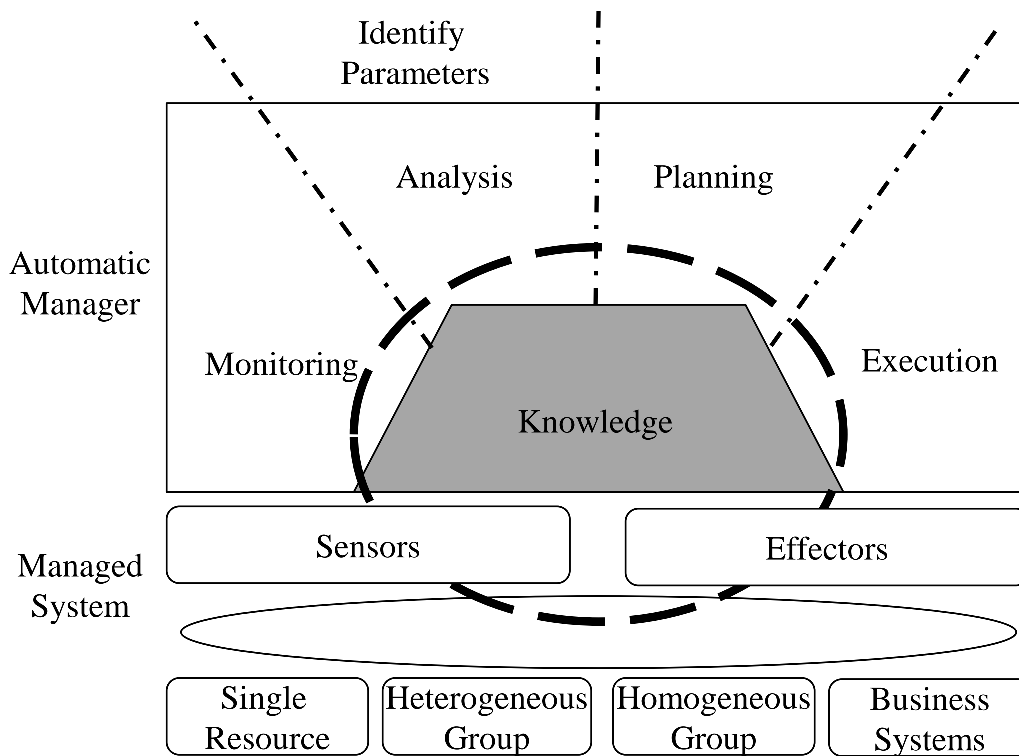


Figure 2.9: Self-tuning Cycle MAPE, adapted from [The05]

***The Analyze Function*:** This function provides the mechanism to observe and analyze current system situation to decide if action must be made. This means correlating and modeling complex situation, which allows predicting performance-related measures based of given parameters.

***The Plan Function*:** This function provides the mechanism to create and select the procedure that will achieve the desired change in the managed system's behavior. The change plan is passed to the execute function.

***The Execute Function*:** This function provides the mechanism to schedule and perform the change plan. This is represented by effectors point in the Figure 2.9. Part of this function could be updating the knowledge-base.

***Knowledge-base*:** Shares the knowledge across the different steps of the process.

2.2.4 (Self-)Tuning Cloud Data Management Systems

The Cloud environment by definition promises self management, which emphasizes the need for tuning and self-tuning solutions. Despite the fact that Cloud storage systems already cover aspects of the tuning process (e.g. load balancing among different nodes), many other issues regarding tuning for different optimization goals remain to be solved.

In the next subsections, we provide an overview of different tradeoffs and optimization goals within the context of Cloud storage systems.

2.2.4.1 Optimization Goals/Opportunities

Optimization goals have the largest impact on system configuration starting from the application layer and ending with the underlying infrastructure. In this section, we provide an overview of the fundamental optimization goals in the context of Cloud storage systems:

***Performance*:** Measured by latency and throughput, performance is crucial for Cloud storage systems. It is affected by several aspects, such as geographical distribution of data, replication, consistency and durability requirements. Overhead caused by events, such as data compaction and scaling the database cluster, also impacts the overall performance of the Cloud data management system. For several applications high latency is considered the same as unavailability and thus minimizing latency, in such case, is a priority.

Availability & Fault Tolerance: In Cloud environment, systems work on clusters of nodes where failure is the normal case, not the exception. Most Cloud storage systems support fault tolerance and recovery. However, even with the promised 99.9% availability, leading Cloud storage providers, such as Amazon, Salesforce.com, and Rackspace, had several outages in the last years causing major websites and businesses to be out of service. Outages result not only in services time-outs but also unrecoverable data losses [DMT11]. In the Amazon's outage of April 2011, data estimated to be 0.07% of the data stored in the US East Region was not recovered.

***Consistency Level*:** For Cloud storage systems, consistency is not a matter of yes or no question. There are several levels of consistency and one can tune it even on the granularity of a single query or data object. Depending on data type, strict consistency is not always a requirement. Since decisions about the level of consistency affect the system performance and availability, it is important to determine the highest achievable level of consistency for specific performance requirements. Trade-offs related to consistency will be discussed in the next subsection.

***Minimum Resource Consumption*:** This goal can be addressed from two perspectives. The first one is regarding *monetary costs*. It is very important to minimize resource consumption, but within specified performance thresholds. The second perspective is *energy efficiency*. Energy consumption of data centers, whether it is for cooling or operating machines, is high and is estimated to increase by 18% every year [ZCB10]. Besides that, up to 35% of this energy consumption is caused by the storage subsystems. Hence, minimizing energy consumption for database clusters is getting more important [BMB12, GLN+12].

2.2.4.2 Tuning Trade-offs

Trade-offs influence the choice of system components of the storage layer, and their configuration. Next, we summarize the tradeoffs for Cloud storage systems. Some of which are typical for any storage system, and some are specific to Cloud storage systems:

***Read Performance versus Write Performance**:* The problem of tuning storage systems for reads versus writes is not specific to Cloud storage systems. However, in the Cloud, challenges related to data replication and distribution add to the problem [SMZ+10].

***Latency versus Durability**:* To store data persistently, it has to be written to disk. However, this results in a high latency. Therefore, many Cloud data storage systems choose to write to memory and synchronize to disk later. This approach lowers latency but could result in data loss in the case of failure.

***Security versus Performance**:* Existing models for supporting security of Cloud data depend mainly on encryption. This affects the performance of the system [WWRL10, Aba09] since it leads to overhead in reading and writing data and increases the overall latency.

***CAP Trade-off Consistency & Availability**:* When a Cloud data management system is optimized for consistency and availability, this implies that no request will work on partial data. In the case of network or partition failure, a request will wait until partitions heal. In this case, the system latency is increased.

***CAP Trade-off Consistency & Partition Tolerance**:* When a Cloud data management system is optimized for consistency and partition tolerance, this means that some requests will work on partial data. Some requests will be refused. In this case, the latency of the system is reduced but its availability as well.

***CAP Trade-off Availability & Partition Tolerance**:* When a Cloud data management system is optimized for availability and partition tolerance, this means that all requests will be answered in all cases. Requests may return inaccurate or stale version of data. In this case, the latency of the system is reduced and its availability is higher.

To tune a Cloud storage system for the previously mentioned optimization goals and trade-offs, the database administrator is faced with a modular system architecture and a distributed environment. The Cloud storage systems' architecture, environment, and requirements put more emphasize on the support of the (self-)tuning process. In the following section, we provide an overview of related work on the (self-)tuning approaches for Cloud storage systems.

2.2.5 Related Work

In this section, we investigate and compare related work on (self-)tuning approaches and frameworks for Cloud storage systems. Related work in this area falls in two parts:

The first one includes tuning Cloud database clusters for specific workloads, optimization goals, or execution environments. Examples of such efforts include the work of [GLN+12], which aims to reduce energy consumption and thus the cooling costs

by applying resource aware data placement and migration strategy. A part of work in this category falls under scheduling. Chi et al. [CMH11] perform cost aware scheduling of queries based on service level agreements whereas Polo et al. [PCC+11] perform Map/Reduce jobs scheduling to maximize resource utilization.

The second part of the (self-)tuning approaches for Cloud storage systems is external to the database system and includes tuning the underlying resources to achieve the optimization goals of the database workload. An example of this is the work of [ASS+09], which focuses on partitioning the CPU capacity of physical machines among different database appliances. A more general example is the work of Xiong et al. [XCZ+11] where they perform cost aware resource management.

The self-tuning frameworks for Cloud storage clusters, include the work of Herodotou et al. [HLL+11]. They developed a framework called Starfish, for self-tuning big data analytics systems and focus on cluster sizing problem. Starfish is build on and designed to work with Hadoop Map/Reduce framework. In other words, it is system dependent.

Reviewing the current state of the art on (self-)tuning Cloud storage systems reveals that there are several diverse solutions for a variety of specific self-tuning issues that fall short of being generalized for other storage systems or tuning goals. There is lack of support for self-tuning frameworks and approaches that are oblivious to the underlying Cloud storage system and that allow tuning for the overall system rather than being specific to certain mechanism e.g. data placement/migration strategy, execution environment or optimization goal.

An essential technique that enables (self-)tuning process is performance modeling. Performance modeling allows predicting systems performance with difference settings and is typically used in what-if analysis to configure tuning knobs. In the coming section, we provide an overview of the performance modeling techniques for database systems.

2.3 Performance Modeling

Performance modeling is defined as: “the process that aims to capture and analyze the dynamic behavior of computer and communication systems” [Hil05]. System’s behavior is expressed by measured performance criteria (such as response time, and throughput), or consumed resources (such as CPU, memory, and energy) in relation to workload or application criteria (such as data size, read and write ratio). Performance models are used to evaluate design options and configuration knobs of the underlying systems for actual or anticipated workload. Such evaluation allows to discover bottlenecks, provision resources, and tune different knobs. In the coming subsections, we will provide an overview of performance modeling for database systems and provide more details about the empirical data-driven approach.

2.3.1 Performance Modeling Approaches for Database Systems

There are basically two approaches for modeling the performance of database systems. We introduce both of them shortly in the coming paragraphs and explain our decision of choosing the second approach.

Analytical Architecture-Level Approach

This approach is based on understanding and analyzing the data storage and retrieval processes [OK12]. To estimate a performance metric of a certain process, the model includes estimation of this performance metrics in the different stages (i.e., different architectural levels of the system), which are required to perform the process. The performance metric of interest is estimated in relation to workload and system parameters. Examples of such parameters are the hit ratio of the database buffer, the probability of lock conflict, the probability that a request is served from disk/memory, etc. [Tho00, YKGS12]. Figure 2.10 includes examples of query processing model based on which a performance model can be built.

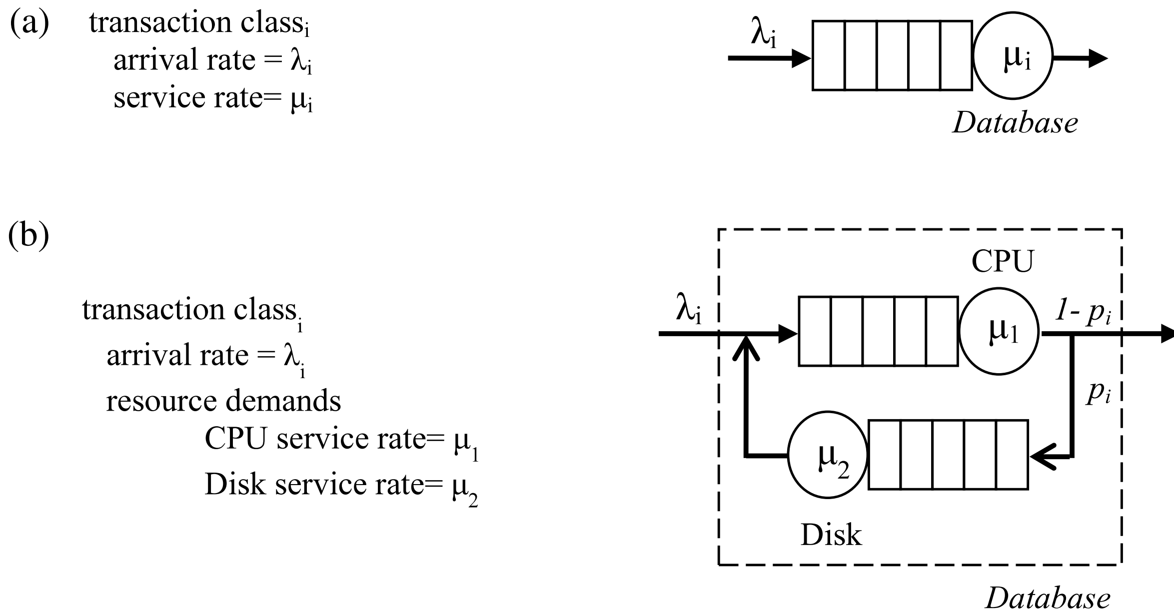


Figure 2.10: Examples of Analytical Models, adapted from [OK12]

Empirical Data-Driven Approach

Empirical driven approach needs experiments to generate data that is used to train a predictive model. Experiments used to generate the data should be representative of the target workload or application scenario for which the system will be tuned later. The work flow of the empirical model is illustrated in Figure 2.11

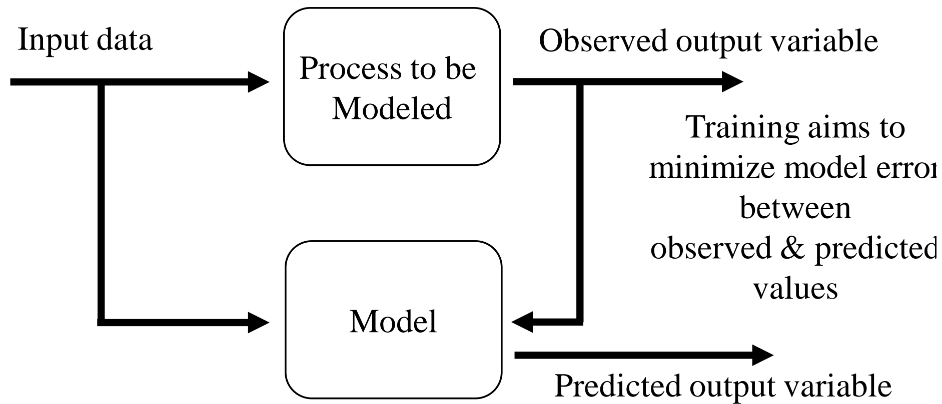


Figure 2.11: Empirical Modeling Approach, adapted from [SSA08]

There are also approaches that merge the two techniques together in what is called semi-analytical performance modeling.

Our choice of predictive modeling approach is driven by our goal of building a database-agnostic framework, thus we avoid using database-dependent knowledge. Sine the analytical approach is tightly coupled with the implementation of the database management system and requires expert knowledge of its internals, we decided to use the empirical model.

We provide more details about the empirical data-driven modeling in the next section.

2.3.2 Data-driven Modeling Approach

A Data-driven modeling approach is composed of two basic steps. The first one is generating statistical data about the performance of the system. The second step is analyzing this statistical data to generate a model.

2.3.2.1 Training Data Generation

Generating statistical data needed for the modeling process involves two things: generating the workloads and collecting performance measurements. There are many tools for monitoring different performance aspects of database systems. The more interesting aspect, in this step, is generating the database workload. We classify existing approaches for generating workloads for database systems' testing purposes into the following:

***Benchmarks*:** Offer a range of workloads which are designed to match different application scenarios e.g., Transaction Processing Performance Council (TPC) benchmarks. Benchmarks are typically DBMS-independent, and support investigating several aspects of the system performance.

***Stress Tools and Load Testing*:** Designed by many database system vendors and bundled with their systems e.g., Cassandra stress tool [The15e]. Because they are system-specific and narrower in scope than benchmarks, they are typically not used for the modeling process.

The system can be monitored continuously and the model could be iteratively refined during the systems runtime to allow self-tuning behavior.

2.3.2.2 Predictive Modeling

A model is defined, in [HMS11], as an abstract high-level global representation of a real-world process. Predictive models are models that allow us to predict the unknown value of a variable of interest, given values of other variables. The aim of the predictive modeling is to find a model (a mapping or a function) that can predict the value of the variable of interest (dependent variable) in relation to a vector of independent variables.

There are several techniques for predictive data-driven modeling that fall under machine learning techniques. Regression analysis is considered the simplest one. It relies on statistical analysis to find a formula or mathematical model to represent the relationship between the dependent and independent variables. Other machine learning techniques include clustering, tree-based, genetic evolutionary algorithms and neural networks [MF10]. These are usually more complicated and costly but more precise than the group of regression techniques.

As stated by [Kee11], there is an infinite number of models that fit an infinite set of data equally well. We will discuss in later chapters our choice of modeling techniques and the resulted models. The next section is dedicated to present a review of existing research on performance modeling for Cloud storage systems.

2.3.3 Related Work

In this section, we provide an overview of related work on performance modeling for Cloud storage systems. Since generating the training data is en essential part for the empirical performance modeling approach, we also provide an overview and a classification of existing Cloud storage systems' benchmarks.

2.3.3.1 Performance Modeling for Cloud Storage Systems

In literature, we could find extensive research on database systems performance modeling [OK12, SMA+08], which is targeted at conventional database systems. There is also large part of work on Cloud applications performance modeling that is not storage system related [CRB+11a, XP09, CRB11b]. Our focus is on the area of Cloud storage systems' performance modeling. In the following we report the related work that falls under this area:

[YKGS12]: This work uses the analytical architecture-level approach for performance modeling. In this work, Yanggratoke et al. present an analytical model of a distributed key-value storage system that is part of the Spotify storage architecture. The response time of the storage system is modeled as a function of the storage system configuration, the load, and model parameters that is measured on the storage servers. The model is evaluated in a test environment and in an operational Spotify storage system. Their tests show high accuracy for lightly loaded storage systems. Based on that, they restrict the applicability of their model to systems containing at most one storage element on average.

As already mentioned this work uses the analytical performance modeling approach, which is storage system specific. A basic need for our framework is the use of storage-system-agnostic performance modeling approach.

[GGK+14]: This work uses the analytical architecture-level approach for performance modeling. In this work, Gandini et al. focus on modeling the performance with regards to the number of nodes and the replication factor. They present two high level queuing network models representing single node and multi-node architectures. In their work, they study the performance of the Yahoo! benchmark [CST+10] with three systems Cassandra, Hbase, and mongoDB. Though they tried to capture the performance of the three systems in two models, the different architectural and design decisions of the three systems lead to different results. The created models capture the throughput and response time (for read and write) of Cassandra and mongoDB with acceptable error rate. However, they have large error rates for Hbase throughput. In their work, they argue that the complexity of the Hbase system makes it impossible to capture its performance in a simple model and get acceptable results.

An overview of related work on performance modeling of Cloud storage systems shows that existing approaches [YKGS12, GGK+14] are analytical and system- architecture dependent. The current state of the art lacks support for system-independent data-driven performance modeling approach, which is important because of its extendability and re-useability.

Generating the training data is essential step for the data-driven performance modeling. In the following section, we provide an overview and classification of existing approaches on benchmarking systems for Cloud storage systems.

2.3.3.2 Benchmarking Cloud Storage Systems

From the early stages of Cloud storage systems, it was argued that the traditional benchmarks (TPC benchmarks) are not sufficient for Cloud storage systems [BKKL09, CST+10]. Traditional benchmarks do not cover Cloud storage systems' properties such as scalability, elasticity, availability, and fail-over. This lead to the development of new benchmarks that are suited for Cloud stoarge systems. We classify related work on Cloud storage benchmarks in three classes: general purpose benchmarks, infrastructure based benchmarks, and feature testing benchmarks.

1. General Purpose Benchmarks:

   General purpose benchmarks include benchmarks, which measure latency for different systems and focus on providing details about selecting workloads and benchmark architecture that corresponds to the Cloud environment and its applications. Such benchmarks are often used as bases for feature testing and infrastructure based benchmarks. Notable work on general purpose benchmarks for Cloud storage systems include the following:

   Bigtable Benchmark [CDG+08]: The Bigtable benchmark was designed to measure the performance and scalability of Google's Bigtable while changing the number of serving nodes. As a performance metric, the benchmark measured the number of written/read values per second. The defined workloads include: sequential write, sequential read, scan, and random read. Though the aim of this work was not to provide a reusable framework and thus was not open-sourced, it is beneficial in defining metrics and means of benchmarking Cloud storage systems.

   Yahoo! Benchmark [CST+10]: This benchmark was developed by Yahoo! researchers to measure the performance criteria of PNUTS [CRS+08] and was extended for other online storage systems (supporting SQL-like queries). This benchmark provides tiers for measuring latency and scalability of the system under test. Workloads are defined by read/write mixes, data size, and request distribution.

   The Yahoo! benchmark gained popularity, and was used and extended by several other benchmarks as we show in the following sections. The reason for its popularity is that it contains a data generator, a workload generator, and drivers for several Cloud storage systems [RGVS+12]. The Yahoo! benchmark was even used to benchmark relational databases [DPCCM13].

   BigBench [GRH+13]: This benchmark is targeted at DBMSs and Map/Reduce systems that promise big data analysis. It reports the response time of the system under test, while testing several workloads. The benchmark workloads are based on the business model of products' retailer. Data and workload generators are developed as part of it. The data generator uses the Parallel Data Generation Framework (PDGF) [FPR12]. The generated data contains structured, semi-structured and unstructured data. The structured concept builds on TPC-DS[1]. The semi-structured and unstructured parts of data capture retailer's website related data, such as users' clicks and products reviews.

2. Infrastructure Based Benchmarks:

   Another group of benchmarks focus on how a system performance changes with different technical, or platform choices. Research efforts that fall under this category include:

[1]Details available at: http://www.tpc.org/tpcds/

[RGVS+12]: Rabl et al. use the Yahoo! benchmark [CST+10] to measure the performance of 6 open-source storage systems of different architectures including: Hbase, Cassandra, Voldemort, Redis, VoltDB, and MySQL. They test the performance with two different hardware setups: memory-bound cluster and disk-bound cluster. In this benchmark, the latency and throughput of the systems under test are measured. The workloads that the benchmark includes are synthetically generated using the Yahoo! benchmark and include a variety of read, write and scan mixes. In the test, the number of requests per workload was scaled along scaling the number of nodes in the serving cluster.

[Apl13]: This benchmark focuses on analyzing the performance of Cassandra on two platforms using HDD or Flash memory. In this benchmark, Aplin et al. present two workloads. The first one they call data ingestion workload. In this workload, data is inserted into Cassandra and then they measure the rate at which data is ingested. The second one, they call data analysis workload. In this workload, data inserted from the first workload is queried. This benchmark includes workloads generated by the Yahoo! benchmark [CST+10] as well. The performance criteria that is covered in this benchmark include latency and throughput.

[BLL+14]: The aim of this benchmark is to compare the performance of Online Transactional Processing (OLTP) application on public Cloud platforms and investigate the effect of CPU performance on response time and total cost of virtual machines. Based on WordPress[2] blogging software, the benchmark introduces an open source workload generator. Workloads are defined based on predefined WordPress tasks such as searching for a keyword, publishing a post, uploading photos. The performance is measured by average response time and total virtual machine cost. In this benchmark, Borhani et al. evaluate different instance types on three Cloud storage platforms: Amazon EC2, Microsoft Azure and Rackspace Cloud.

3. Feature Testing Benchmarks:

   The last group of benchmarks examine Cloud-related properties of storage systems, such as scalability, elasticity, replication, and consistency levels. Research efforts that fall under this category include the following:

[DMRT11]: In addition to performance measured by latency, this benchmark provides measures for elasticity and scalability. This benchmark's workloads and data are based on the Wikipedia use case. Workloads represent users asking, in parallel, to read and update articles. The data set used is based on Wikipedia dump. In this benchmark, the performance is defined by the time needed to complete a given number of requests. The measurements are not focused on the average response time but on total time needed to complete a

[2]WordPress available at: https://wordpress.com/

number of requests. The scalability is defined as the change in performance when nodes are added and fully integrated in the cluster. The elasticity is measured by two metrics. The first one is the time needed for the cluster to stabilize. The second is the impact on performance when nodes are added or removed. The benchmark reports results on three Cloud storage systems, namely: Cassandra, HBase, mongoDB.

[SMZ+10]: The aim of this benchmark is to measure scalability, speedup, and fault tolerance of Cloud storage systems. This benchmark is based on [CDG+08, PPR+09]. In their work, Shi et al. design experiments to measure scalability and speedup of Cloud storage systems including Hbase, Cassandra, HadoopDB, and Hive. The workloads and tasks are derived from Google's Bigtable benchmark [CDG+08] and [PPR+09]. They divide their work into two benchmark tiers. The first one is a read and write benchmark, which includes workloads of sequential and random read, write, and scan mixes. The second tier is structured query benchmark tier. In this tier, they define workloads that include aggregate operations. The scalability is characterized by the change in throughput when increasing the number of nodes. Shi et al. introduce a base time as the elapsed time of a system with cluster size 5. Then, they define speedup measure as the ratio between the elapsed time of a task with cluster size k, and the base time.

[WLZZ14]: The purpose of this benchmark is to monitor how performance changes when changing the replication factor and the consistency level. Several benchmarks that differ in data size and purpose are defined. The workloads in these benchmarks are based on the Yahoo! benchmark [CST+10] and are defined as the following: read mostly, read latest, read and update, read-modify-write, scan short range. This benchmark reports results on two systems: Cassandra and Hbase.

[KKR14]: Based on the Yahoo! benchmark [CST+10] and the work [RGVS+12] by Rabl et al. In this benchmark, Kuhlenkamp et al. measure scalability and elasticity of two Cloud storage systems, namely Cassandra and Hbase. In their work, scalability is measured by how much performance changes when resource capacity is changed. Elasticity is measured by how efficient the system can scale at runtime in terms of the scaling speed and its impact on performance. In this work, Kuhlenkamp et al. discuss the reproducibility of benchmarking result on physical and virtual infrastructure. They extend the work of Rabl et al. [RGVS+12] by benchmarking Cassandra on variety of Amazon's Elastic Compute Cloud (EC2) setups with different instance types and storage setups. They investigate the concepts of vertical and horizontal scaling. In *horizontal scaling*, they add/remove nodes of the same instance type. In the case of *vertical scaling*, they stop an instance, increase its allocated resources by changing its type, and restart the instance again. The workloads that are used in this benchmark are generated using the Yahoo! benchmark.

[PS15]: In this work, Pokluda et al. benchmark fail-over characteristics of large scale data storage systems. They use the Yahoo! benchmark [CST+10] to generate workloads. Performance measurements focus on throughput, latency, and the number of reported errors while simulating node failures. A node failure is performed in this benchmark by stopping a cluster node and bringing it online again after a specified time.

In Table 2.1, we provide a comparison of the surveyed Cloud benchmarking systems. For the systems that use another work to generate their workloads and data, we use "-" for the extendability field, since they are an extension of another work.

The existing benchmarks are designed for comparing the performance of several database systems for certain workloads or checking the performance of one database systems with different workloads. They also include measures for different Cloud-properties. Generating the training data for performance modeling requires automating the testing of the system with different workloads and different configurations. The existing benchmarks can not be used as a training component without further modifications. Later in the thesis, we illustrate our design of a training component that can be used as a Cloud storage systems benchmark and automates the testing process with different cluster configurations and different workloads.

Benchmark	Focus	Workload	Data	Extendable
[CDG+08]	general	synthetic	synthetic	no
[CST+10]	general	synthetic	synthetic	yes
[GRH+13]	general	product retailer	synthetic	yes
[RGVS+12]	different infra.	uses [CST+10]	synthetic	-
[Apl13]	different infra.	synthetic, [CST+10]	synthetic	no
[BLL+14]	different infra.	Wordpress	Wordpress	yes
[DMRT11]	Cloud feature	Wikipedia	Wikipedia	yes
[SMZ+10]	Cloud feature	based on [CDG+08]	synthetic	yes
[WLZZ14]	Cloud feature	uses [CST+10]	synthetic	-
[KKR14]	Cloud feature	uses [RGVS+12, CST+10]	synthetic	-
[PS15]	Cloud feature	uses [CST+10]	synthetic	-

Table 2.1: Comparison of Related Work on Benchmarking Cloud Storage Systems

2.4 Summary

In the previous sections, we described the properties of the Cloud environment and its storage requirements. These properties enticed many companies to host their applications and services solely on the internet. The storage requirements of Cloud applications

could not be fulfilled with traditional DBMSs. These requirements, such as scalability, elasticity, fault tolerance, high availability, and evolving schema lead to the design of new breed of storage systems and services. We classified these systems into two basic categories. The first one is relational DBMSs as a service, which include services such as Amazon RDS and Microsoft Azure. The second category includes NoSQL storage systems. There is a wide range of storage systems and services that fall under the NoSQL category and vary between distributed file systems and fully-fledged data management systems. We provided a classification of these systems based on different aspects: architecture, cluster management, data partitioning, data replication, data models, data access, CAP theorem, and consistency level.

Then, we presented the tuning reference architecture for database systems. This architecture defines the different layers and tuning possibilities that are fundamental for a successful database tuning process. The self-tuning of database systems is described by the IBM's MAPE paradigm. After providing an overview of different steps of the MAPE cycle, we surveyed fundamental optimization goals and tradeoffs in the context of (self-)tuning of Cloud storage systems. We also provided an overview of related work on (self-)tuning approaches and frameworks for Cloud storage systems. This overview shows that there is a lack of support for self-tuning frameworks and approaches that are independent of the underlying Cloud storage system and that allow tuning for the overall system rather than being specific to certain mechanism, goal, or workload.

Last, we presented performance modeling for database systems. Database performance modeling includes two basic approaches. The first one is the analytical architectural-level approach, which is system-specific. The second is the empirical data-driven approach, which is system-agnostic. Related work on performance modeling for Cloud storage systems lacks support for the system-agnostic, re-useable and extendable empirical approach. At the end, we provided an overview and classification of extincting benchmarking systems for Cloud storage systems and explained our decision of building a training component as part of a (self-)tuning framework as we present in the following chapter.

3. (Self-)Tuning Cloud Storage Clusters

This chapter shares material with CLOSER13 paper "Clustering the Cloud: A Model for (Self-)Tuning of Cloud Data Management Systems" [MSB13] and the ADBIS15 paper "A Self-Tuning Framework for Cloud Storage Clusters" [MSS15]

The goal of this thesis is to provide a database-agnostic, platform-independent, and extendable self-tuning framework that can be used as an adviser for tuning Cloud database clusters, serving one or several applications with divergent and possibly competing workloads. In this chapter, we present the design and basic concepts of our approach in building this framework.

At first, we provide an overview of the concepts for database (self-)tuning in the context of the Cloud. Then, we present a formal definition of the tuning-problem in our scenario. Afterwards, we introduce the basic architecture of our self-tuning framework. Then, we discuss the mechanism and concepts of its components. At the end of this chapter, we discuss the different modes for operating the framework and their differences.

3.1 Overview and Scope

A fully efficient and successful (self-)tuning process requires knowledge about different system components, applications and working platforms. While (self-)tuning in each of these involves fixing local problems, it also involves global coordination between them. As a reference for the (self-)tuning process in Cloud storage clusters, we provide the following concepts:

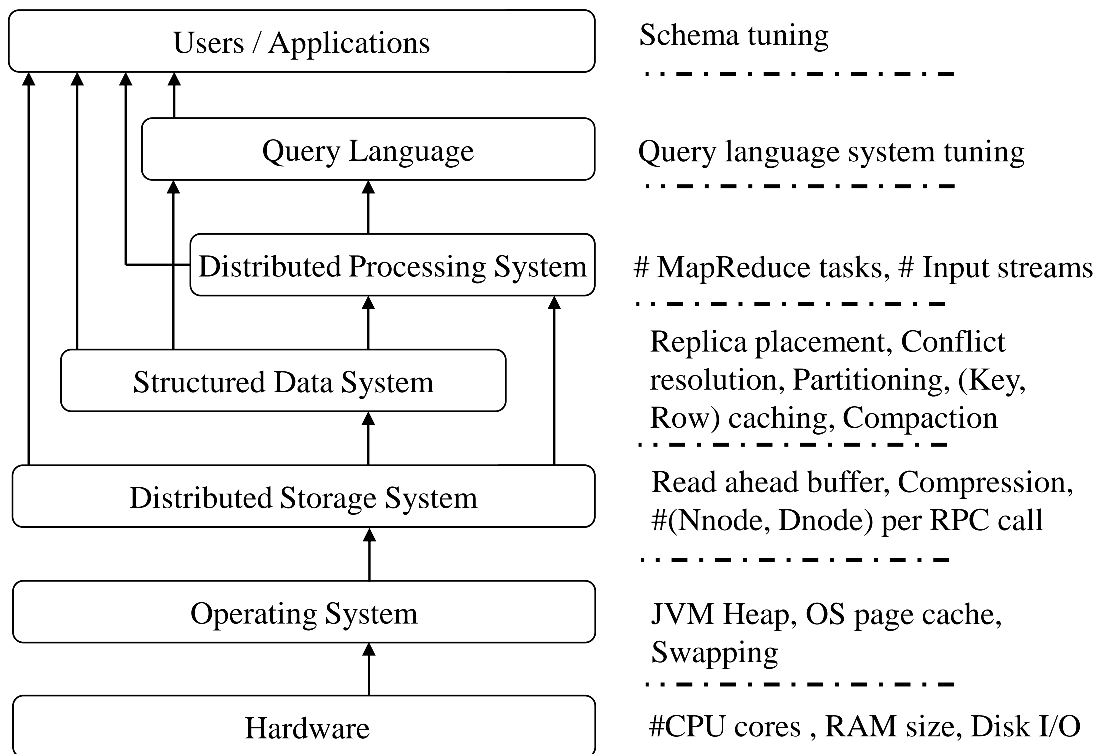


Figure 3.1: (Self-)Tuning Reference Architecture for Cloud Storage Systems

(Self-)Tuning Reference Architecture

Based on the Cloud storage system modular architecture in Section 2.1.4.1 on page 9 and the database tuning reference architecture in Section 2.2.2.2 on page 20, we present the (self-)tuning reference architecture for Cloud storage systems. This architecture provides the blueprint for the (self-)tuning process, and elevates the delivery of a re-useable and extendable self-tuning solutions.

We illustrate the (self-)tuning reference architecture in Figure 3.1. This figure presents the different layers that a system administrator or a tuning framework should take into consideration during the tuning process. Examples of possible tuning knobs, in each layer, are given on the right side of the figure. We discuss them, in details, in the following.

As already mentioned, the tuning process starts with a deep understanding of ***application requirements*** to identify thresholds and optimization goals. Important requirements regarding Cloud applications include elasticity (scaling up and down), consistency (strong to weak), high availability, throughput, and latency. These requirement translate to tuning knobs on the different layers. Possible tuning knobs in the application layer include:

- CAP tuning: e.g., increasing and decreasing the consistency level.

- Query tuning: e.g., rewriting queries.
- Schema tuning: e.g., key-space merging or partitioning.

The tuning process also requires a deep knowledge of ***the storage systems' architecture and features***. In the Cloud, the storage system architecture itself is flexible and can have different components combinations. Based on our proposed Cloud storage systems architecture, the tuning of Cloud storage systems includes the following:

- Query language system: e.g., among several systems that support SQL-like query languages, options like using in-memory processing and data pipe-lining can be tuned.
- Distributed processing system: e.g., tuning the number of Map/Reduce tasks, or input streams.
- Structured storage system: e.g., tuning the replica placement strategy, conflict resolution, and partitioning technique.
- Distributed storage system: e.g., tuning the number of data and name nodes, read ahead buffer, compression techniques.

Furthermore, the tuning process involves ***the operating system and hardware layers***. When deploying the storage system on an in-house or a private Cloud that is dedicated for one specific application, tuning operating system and hardware layers is very much similar to those when working with traditional data storage. It deals with concepts, such as Java Virtual Machine (JVM) settings, page caching and swapping, CPU cores, storage and memory size, etc.
However, when the database is out-sourced to a public Cloud environment, these different hardware and software settings are packaged under different instance types and sizes set by the Cloud service provider. This results in an alternative scheme or arrangement of tuning and optimization alternatives, such as Amazon's micro, small, medium, large, and xlarge instances optimized for general purpose, storage, and computation, etc. The tuning decision, in such case, is abstracted to changing instance type and size, etc. In this case, and because of Cloud environment characteristics of virtualization and multi-tenancy new problems, such as noisy neighbor syndrome, which we discuss later in this chapter, arise.

Based on this reference architecture, application requirements are mapped to different tuning knobs to achieve its optimization goals. In the multi-layered, modular architecture and distributed environment, these requirements can now be handled on two dimensions as illustrated in Figure 3.2 on the next page.

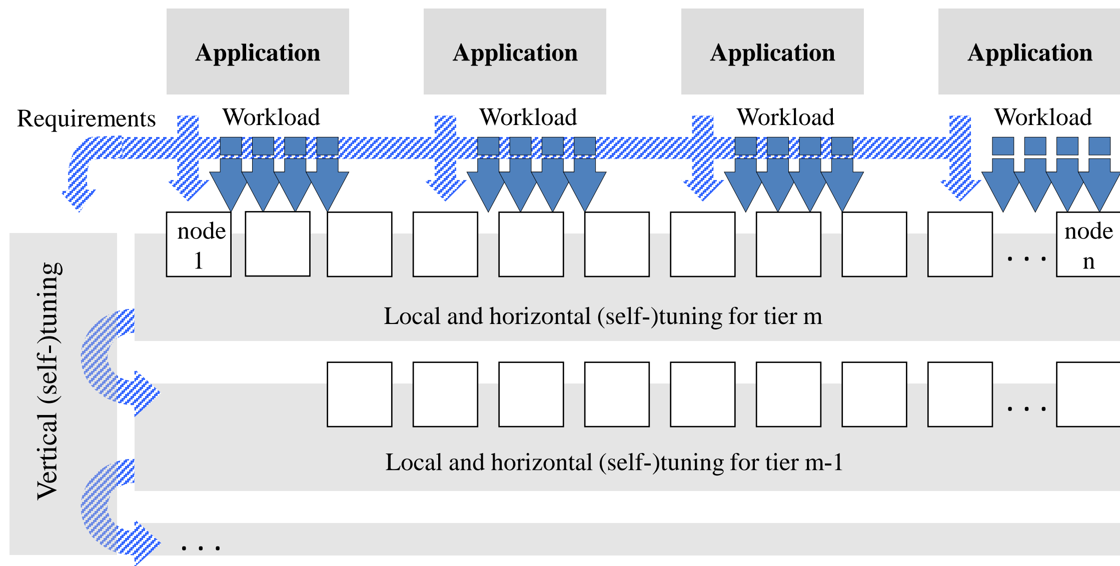


Figure 3.2: Illustration of the Complexity of Self-tuning Multi-layered Cloud Storage Systems

Horizontal (Self-) Tuning

Horizontal (self-)tuning takes place within one layer and across distributed cluster nodes. It includes aspects such as partitioning, load balancing, replication, update strategies, and automatic scaling, etc. Problems in this dimension are better supported, in contrast to the second dimension, because of the homogeneous processes of typically one component type within one layer. However, real-life scenarios impose heterogeneous nodes/components within one layer, which leads to further complications.

Vertical (Self-) Tuning

Vertical (self-)tuning is carried out across layers. It includes the mapping of application requirements expressed as optimization goals, service levels, and thresholds, etc. to specific tuning knobs on each level of the storage architecture. Decisions made on a layer affects the space of tuning knobs and decisions made on the other layers.

Because of the typical shared nothing architectures with data partitioning and replication, some performance aspects can be easily addressed for the overall system. Nevertheless, the typical multi-layered distributed architecture of several component systems adds complexity to the tuning tasks. Moreover, if there are several applications with different and possibly changing requirements, using the same data storage cluster, there is little chance to tune for a specific application. Within the aforementioned scenario, we define the tuning-problem in the following section.

3.2 A Framework for Self-tuning Cloud Storage Clusters

In this section, we formalize the problem of (self-)tuning for a Cloud storage cluster serving one application with divergent workloads or several applications with different workloads and thus optimization goals. Then, we go into details describing the framework architecture.

3.2.1 General Problem Statement

For our framework, we define the general optimization approach as follows: the optimization goal *opt* is to find a Cloud storage system cluster configuration c out of a set of possible configurations CC that minimizes (assuming a standard form of the problem) the aggregated costs for all workloads w of a set of workloads WL that need to be supported by the overall cluster while still fulfilling their performance thresholds.

$$opt = \underset{c \in CC}{\text{minimize}} \quad \underset{w \in WL}{\Gamma} cost(w, c_w)$$

Here, Gamma Γ represents an aggregation function suitable to the given cost components and the considered optimization goals, e.g., average or maximum for response time, sum for energy consumption or latency, such as:

$$opt = \underset{c \in CC}{\text{minimize}} \quad \sum_{w \in WL} latency(w, c_w)$$

Constraints regarding fulfilling performance thresholds can be expressed explicitly or derived from service level agreement and are used to define the cluster configuration search space as discussed later in Section 3.2.6.1. In the following we categorize and provide details of the variables that can be included in the model:

Cluster Configurations in CC: These independent variables are controlled variables and represent the actual tuning knobs that can be used to achieve the optimization goal. Typical configuration aspects are for instance the cluster size, hardware being used, replication factor and other database parameters, etc. Formally, c can be described as an n-tuple that holds relevant parameters as components, e.g., $c_1 = \{clusterSize = 10, replicationFactor = 3\}$ for a data storage cluster deployed on infrastructure of 10 computing nodes and using data replication factor of 3.

Workload characteristics in WL: These independent variables describe the application, but are not controlled by the systems' administrators or developers, i.e. though they may be highly dynamic, they can not be changed deliberately to achieve an optimization goal. These workload characteristics include criteria,

such as access frequencies, user numbers, data volume and schema, etc., which, again, can be modeled as an n-tuple, e.g., $w_1 = \{readPer = 90, writePer = 10, dataSize = 10, CFNum = 5\}$ for a workload having 90% read operations performing on a 10GB database of 5 column families.

Optimization Goal *opt***:** The dependent variables used in prediction models for system optimization are typically those, for which an optimal value should be achieved. For Cloud storage, these may include variables, such as throughput, latency, energy consumption, resource utilization, and consistency, etc. The optimization task may be multi-objective, and in some cases the objectives could be contradicting. An example of such case is optimizing the system for minimal latency and minimal monetary cost (or used resources). In this case, it is possible to assign different weights to the objectives making one of higher priority than the other.

Not all of the possible parameters describing a workload or a cluster configuration may be relevant or desirable to consider in a given application scenario. While we discuss techniques to create a performance model, here it is not our intention to investigate the complex space of variables and their dependencies in its entirety, but rather focus on – in our opinion – a most relevant subspace , namely finding the optimal size of sub-clusters for a given set of workloads. This will be the focus of our implementation in later chapter. In the coming section, we discuss the relevance of the cluster size problem and present the optimization problem for clustering the Cloud, in that context.

3.2.2 Clustering the Cloud

In the case when several applications are competing on resources in one cluster as could be in Figure 3.2 on page 40, it is not possible to make decision for the tuning process when these applications have different workload characteristics and thus different optimization goals and performance thresholds. Identifying workload characteristics is an established research field [CS93, CMT00] and in recent years, some Cloud-specific research was done [KYTA12, MHCD10]. In our work, we will concentrate on the tuning process assuming input as different sets of identified workload characteristics, where each set contains one or more workload with the same optimization goal.

The essential and first step in the tuning process is then to identify sub-clusters size to be able to move from Figure 3.2 on page 40 to Figure 3.3 on the facing page and tune the cluster for the common optimization goal. To achieve this, we express the relation between performance metrics of a workload w with cluster configuration parameter namely cluster size n_w as a cost function; where N is the total number of nodes in the infrastructure:

$$opt = \sum_{w \in WL} cost(w, n_w) \rightarrow min$$

subject to constrain

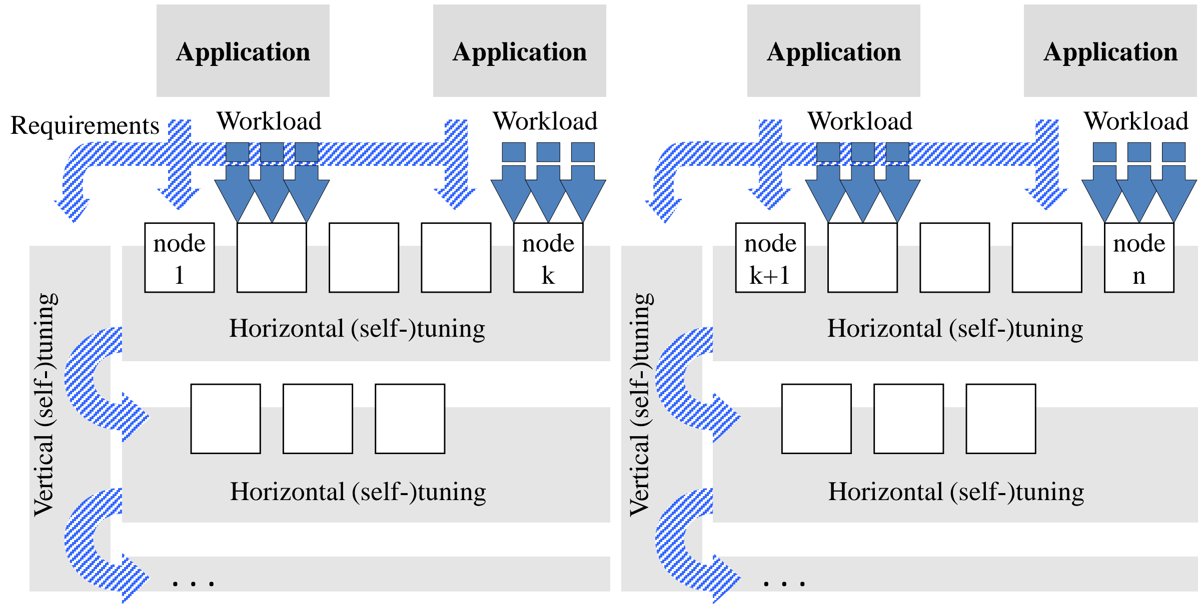


Figure 3.3: Divide and Tune

$$\sum_{w \in WL} n_w \leq N$$

The goal is to partition N nodes among set of workloads WL. The cluster size directly affects the data size and the request throughput that the storage cluster can handle; where more nodes does not necessarily achieve a better performance since increasing the cluster size has a counter effect on the performance of storage system because of data distribution and replication. This will be discussed later in this chapter. Based on the previous assumption our approach applies the concept of *Divide and Conquer* by creating dedicated sub-clusters for single applications/workloads or groups of workloads with similar requirements as shown Figure 3.3.

3.2.3 Framework Design

We design the architecture for our framework based on *the General Process of System Tuning* in Section 2.2.1 on page 18 and *the Database Self-tuning MAPE Cycle* [The05] in Section 2.2.3.2 on page 22. First, we start by defining the input for the framework, based on the constrains defined in *the General Problem Statement* in Section 3.2.1 on page 41, as the following:

- Workload characteristics WL.
- Optimization goals *opt* and thresholds.

The output of the framework is pairs of workload and the cluster configuration that achieves workload's requirements $<WL, c_w>$. Cluster configuration set CC includes:

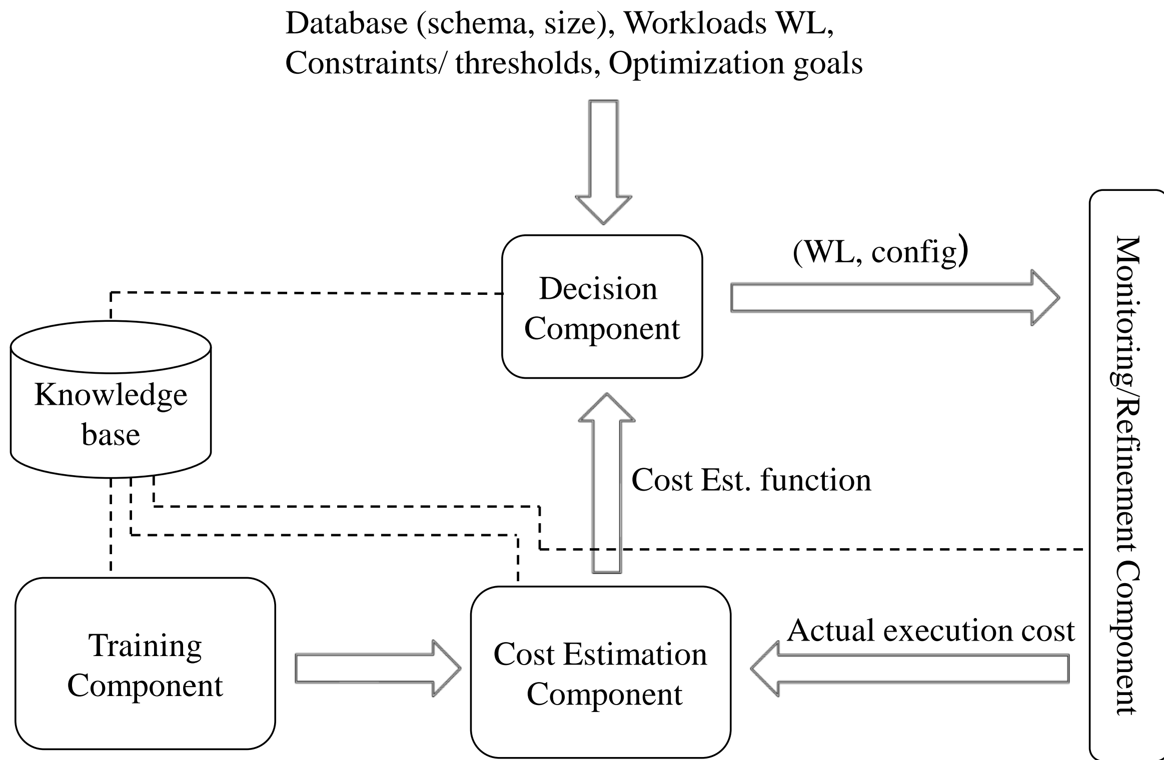


Figure 3.4: Self-Tuning Framework for Cloud Data Management Systems

- Infrastructure configuration (both hardware and software).
- Database system configuration.

More details about input and output parameters are delivered in the coming sections along with corresponding components.

Identifying a suitable configuration is a combinatorial optimization problem in which the measure for evaluating the "goodness" of different alternatives depends on predicting the performance of the data storage cluster. For performance prediction we depend on analyzing historical performance measurement, which can be extracted from monitoring the database system cluster during a training phase or during runtime application. To complete the MAPE cycle, we use a feedback loop that allows continuous refinement of the prediction model.

Based on the previous, we designed our framework from the following components, which are illustrated in Figure 3.4:

Training Component: The purpose of this component is to generate the training data, which is needed to model the performance of data storage cluster for a certain workload with different cluster sizes or possibly different configurations.

Cost Estimation Component. This component uses statistical-based data-driven modeling to build performance models as mathematical functions. These functions are derived by regression techniques done on statistical data gathered from the training phase.

Decision Component: Tuning knobs are expressed as different independent variables during the modeling process. Based on conditions derived from workload thresholds, this component performs a filtering process on the value-space of the independent variables. Then, it solves the optimization problem of the cost models, based on the optimization goals of different workloads, to find preferable values of the tuning knobs.

Monitoring/Refinement Component: This component is responsible for adding measurement to the knowledge-base and initiating a re-modeling process when the prediction model fails to provide the required accuracy.

Knowledge-base: Stores information for re-use by the framework components. Information includes workloads description (i.e. schema and data access pattern), cost models, and performance measurements .

The monitor function of the MAPE cycle is fulfilled by the framework's training component in the offline phase. During the online phase, this function is fulfilled by the framework's monitoring/refinement component. MAPE's next function, analyze, is fulfilled by the cost estimation component, which analyzes the collected data from the monitor function to generate cost estimation models. Whereas, the plan/execute functions are fulfilled by the decision component based on the generated models.

3.2.4 Training Component

The purpose of our training component is to generate data (i.e. statistics regarding systems performance with different workloads and different configurations), that is needed to model the performance of a Cloud's storage cluster. The training component is essential to enable the framework's offline phase. Since our general goal is to find the cluster configuration that best achieves the optimization goals of different workloads, the training component should consider different aspects of the cluster's performance with different configurations and different workloads. The training phase starts by generating the workloads. Existing tools, which are typically used for generating workloads for test purposes include stress-tools and benchmarks.

Stress-tools are bundled with data management systems so they can not be used as a building block for a framework that needs to be data management system agnostic.

A variety of benchmarks for Cloud storage systems were developed [CST+10, VKJ14, KKR14, WLZZ14]. We provided a survey and a classification of these systems in Section 2.3.3.2. As already discussed, existing systems do not support the automation of

testing Cloud storage systems with different infrastructure/workload settings. For this reason, we decided to build our own training component.

Our training component should support the typical requirements of a benchmark, such as allowing workload configuration: read/write ratio, data size, throughput, etc. It should also automate the testing process with different cluster configuration. As already stated, the testing for an increasing number of database system cluster size is an important aspect and should be automatically supported by the training component.

Though our training component is designed with different goals than a benchmark, it can be used as a benchmark. In our implementation in Chapter 4, we measure only latency. Compared to the classification provided in Section 2.3.3.2, our training component falls under both general purpose and infrastructure-based benchmarks.

The means for measuring consumed resources and performance metrics, differs from one cost to another. In the case of latency, response time and CPU cores or memory, it is straightforward. Operating system and DBMS commands for monitoring such performance metrics are widely available. Other costs, such as monetary costs or energy consumption require additional equipments or additional information from the Cloud storage service provider.

Supporting monetary cost estimation model, requires additional information from the Cloud storage service provider. The Cloud storage providers introduce different price plans and different granularities of the pricing, which requires additional work to introduce a unified comparable pricing. In the case of an open source data storage system installed on private clusters in-house, this requires more effort from the framework user, setting up a price per storage-unit list for her/his storage infrastructure.

For electricity consumption model, special equipments, such as wattsup[1] for measuring energy consumption of the computing nodes under workload, are typically used. The generation of electricity consumption estimation model depends on the availability of such tools.

In our implementation, which we present in next chapter, we focus on latency, though other aspects, such as throughput, can easily be considered.

3.2.4.1 Training Component Architecture

We provide an overview of the training component architecture in Figure 3.5 on the facing page. The essential part of the this architecture is the training component manager, which is responsible for starting the training component instances. It also works as database cluster controller, performing operations of:

[1]Information about wattsup can be found in: https://www.wattsupmeters.com/secure/products.php?pn=19

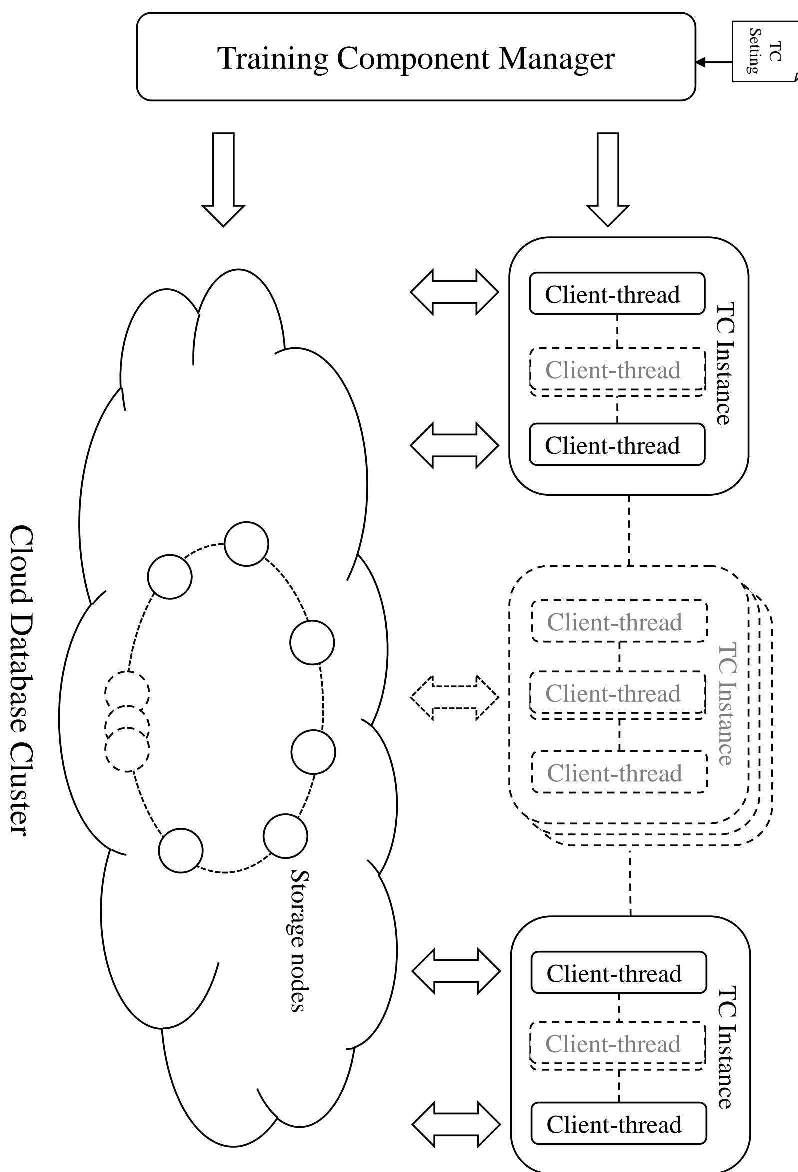


Figure 3.5: Training Component Architecture

- ***Setting up the database system***: This involves starting the database system cluster with different configurations. Configuration includes DBMS settings, such as replication factor and replica placement strategy. It also includes the underlying infrastructure soft- and hard- configuration, such as the JVM heap size and the number of nodes in the cluster.
- ***Preparing the database system for the training workloads***: This involves creating the database schema. Loading training data, or generating and loading data before the actual training workload, if needed.
- ***Clearing caches in the case of a cold run***: This is achieved by rebooting database and operating system to flush the file system caches, main memory and CPU caches.

The settings for the training component controller are provided by the framework user in an Extensible Markup Language (XML) file beforehand and include the training workloads characteristics as well. Depending on information about the expected/targeted workloads and available resources, the framework user sets the training component settings. The schema for the training phase settings file is illustrated in Figure 3.6 on the next page and Listing A.1 on page 106. The training component manager starts a number of training component instances. The training component instances are responsible for starting the training workloads and collecting measurements of performance. We designed the training component to allow specifying the following workload characteristics:

- Cluster and node settings: storage path, nodes Internet Protocol (IP) address list, data placement strategy, etc.
- Database schema (tables, number of columns), record size and replication factor, etc.
- Data access specifications: read and write ratio, number of rows to be read or written, throughput (number of concurrent access), etc.
- Training component instance configuration such as the number of clients and threads, etc.

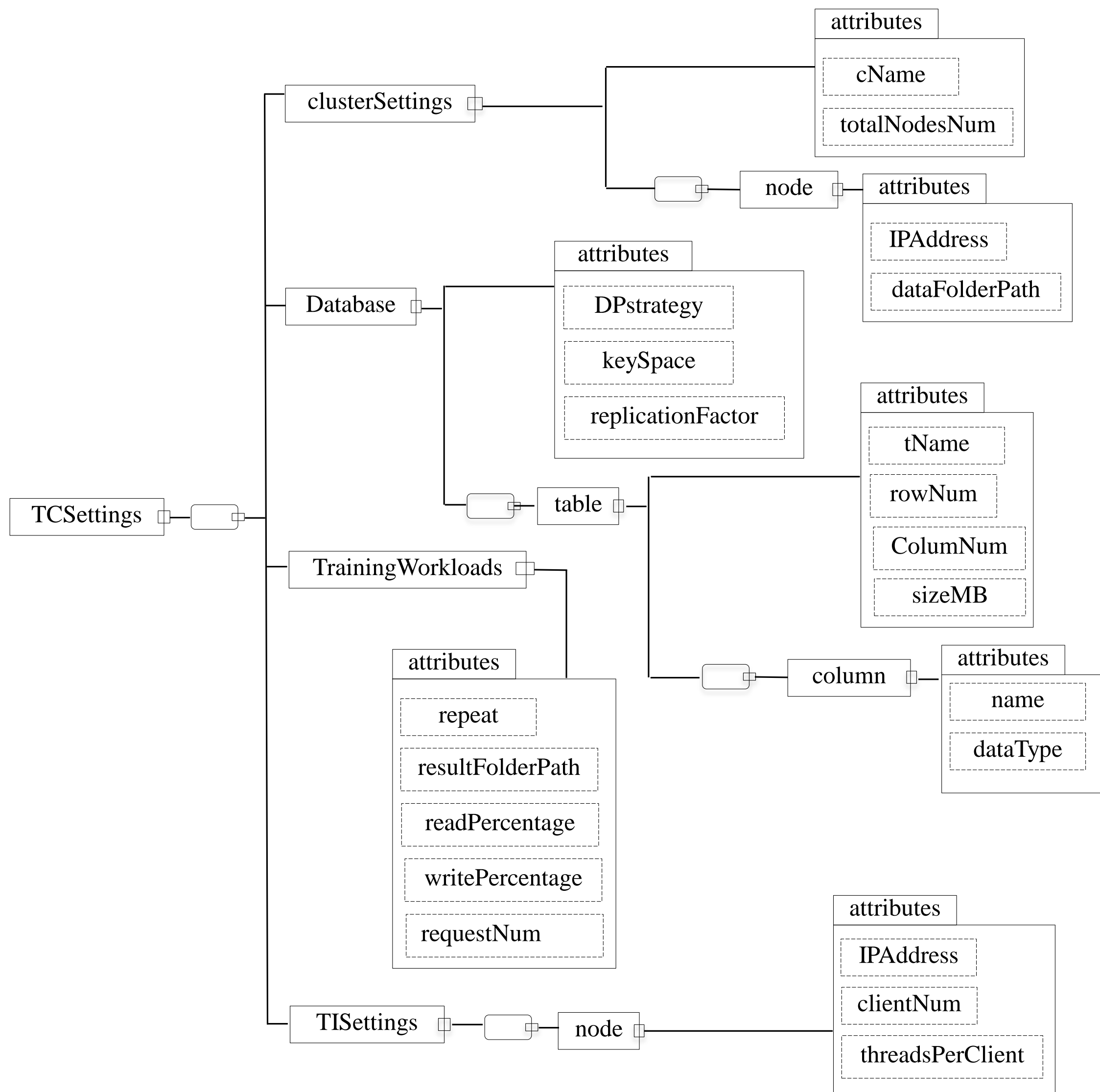


Figure 3.6: The XML schema for the Training Component Settings

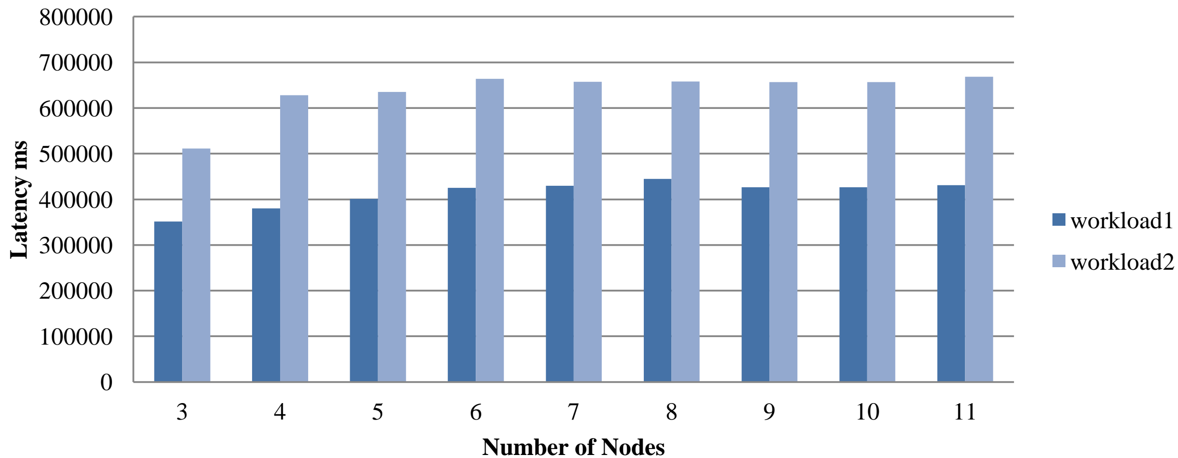


Figure 3.7: The Performance of Underloaded Cassandra Cluster

3.2.4.2 The Training Phase

As specified in the training component settings file Listing A.2 on page 108 and based on the available infrastructure, several phases of the training process are performed. Each phase is defined by an iteration of the training component settings with different values for the configuration parameters that need to be modeled. The value space of each parameter is created based on database systems settings and available infrastructure/physical resources.

A training phase starts by building the data storage cluster on the available infrastructure. In each phase, multiple remote training component instances are started by the training component manager using SSH (Secure Socket Shell). Each training component instance starts multiple client-threads depending on the number of the CPU cores and the memory size of the host machine. The client-threads generate data access requests achieving the targeted workload. After the workload ends, statistical data describing the performance are retrieved from all client-generator machines and combined in one output file. Within one phase, the training component automatically repeats the experiment a number of times defined by the user and the average measurement is used for the modeling process. After the experiment is done for the current configuration, the training starts again for next iteration of the training component settings.

- Deployment on Virtualized Infrastructure: Like many other systems deployed on multi-tenant visualized architectures, Cloud storage systems face the noisy neighbor problem. This problem was reported on several Cloud hosting platforms (e.g. noisy neighbor problem is considered one of the top five performance problems on as Amazon AWS). Degradation in performance happens when other tenants steal physical server resources, such as CPU or network bandwidth. This results in fluctuations and unpredictability of the performance on such platforms, making it unideal for benchmarking or performance modeling. However, a case can be made if the real deployment will be made on the virtual environment as well. The

effect of noisy neighbors can be minimized by repeating the experiments in the case of benchmarking.

- Workloads Causing Under-load Condition (throughput and data size): Cloud storage systems were made to serve hundreds of thousands of requests concurrently. Unless the generated workload reaches high concurrency, the performance of a Cloud storage system will look poor compared to conventional centralized DBMSs. Moreover, the performance of the system will not improve when increasing the number of serving nodes as meant to be with Cloud database clusters. Figure 3.7 on the facing page illustrates the result of several experiments we conducted on a Cassandra cluster(more about it will be explained in the next chapters). We display the average latency of Cassandra with two different workloads each of which with different cluster sizes. Even with increasing the cluster size, we notice that the performance gets worse and slightly improves at few points. This is due to the fact that with small workload, the overhead of distributing and replicating data to new nodes is more than the gain in performance. This should be part of the model to identify the point when increasing the number of nodes in the underlying cluster does not improve the performance and has a counter effect.

3.2.5 Cost Estimation Component

We use cost in our work to refer to consumed resources or performance metrics whose minimization and maximization can be used as an optimization goal to lead the tuning process. Here, the cost can be memory footprint, energy consumption, or monetary cost. Most likely it is response time, latency, and throughput. We refer to a model that enables the prediction of a performance metric as a cost estimation model. The cost estimation component is responsible for generating such a model from historical data, which can be generated by the training component and/or the monitoring/refinement component. The needs of our framework lead to the following requirements that must be fulfilled by the cost estimation component:

- Integrate different costs, which allows for multi-objective optimization. For example, a workload optimization goal is minimum energy consumption with maximum latency.
- Allow continuous cost model refinement: this is enabled by the monitoring component, which collects new measurements that are used to update the model.

As a method for generating models we use statistical-based data-driven modeling techniques [SSA08] in developing mathematical functions that represent the relationship between performance metrics on the one side as dependent variables and the workload characteristics and configuration knobs as independent variables on the other side. We define the input for the cost estimation component as the following:

- Measurements: This includes performance metrics from running test on the current infrastructure including the Cloud storage system. This data is generated by the training component and define the dependent variables.
- Workload Characteristics: This information is input from the system administrator or tuner and includes criteria, such as the type of workload identified by the read and write ratio, data size, etc. It defines the independent variables.
- Cluster Configuration: This includes infrastructure characteristics that will be used as tuning knobs, such as the number of computing nodes (cluster size). Other criteria, such as database management system configuration can also be used, here.

For each performance metric, an estimation model is generated. It is possible to use several models for different performance metrics for a certain workload depending on the tuning goals of that workload. Based on [SMK+11], since generated models are refined online, a model would then be able to correct itself if the initial training models are not representative of the running workload.

3.2.5.1 Model Generation

To build a model that fits the data that is generated in the training phase, we propose the use of machine learning techniques [HMS11]. In the implementation chapter, we list the considered techniques, discuss and evaluate our choice of using regression analysis.

The data we use as input for the modeling process is of the form in Table 3.1. 90% of this data is used as training set to generate models. The measurement field contains values of e.g., latency, throughput. The workload characteristics contain values for: percentage of the write requests in the workload wr, percentage of read requests r (random reads, sequential reads(scan), single-row write, batch write, update, etc.. can be used), s is for data size and nc is for the number of tables (column families). Whereas the cluster configuration contains values for cluster size cs, and data replication factor rf, etc.

Measurement	Workload Characteristics	Cluster Configuration
value	$w_1 =< wr, r, s, nc, ..>$	$c_1 =< cs, rf, ..>$
value	$w_1 =< wr, r, s, nc, ..>$	$c_2 =< cs, rf, ..>$
value	$w_2 =< wr, r, s, nc, ..>$	$c_1 =< cs, rf, ..>$
....	....	....

Table 3.1: Training Data as Input for the Modeling Process

The framework can be extended by deploying different machine learning techniques. Depending on the number of dependent variables and the size of the training data set, one of the techniques will yield the best model (among the other generated by the framework) each time. In the next section, we introduce how the framework chooses a model among the multiple ones generated in the modeling process.

3.2.5.2 Model Selection

To perform model selection, we use the remaining part of the data set (10%) as a test set. After generating the models, the framework chooses one based on the accuracy of their prediction against the test set.

For our framework, we examine the prediction power of the generated models against the test data set. For that we use the Mean Absolute Percentage Deviation (MAPD) [APB15], also known as the Mean Absolute Percentage Error (MAPE). MAPD is a measure of the prediction accuracy that is typically used for evaluating forecasting estimation methods. It expresses the accuracy as a percentage, and is defined by the formula:

$$MAPD = \frac{1}{m}\sum_{i=1}^{m}\left|\frac{Actual_i - Predicted_i}{Actual_i}\right|$$

where m is the size of the test data set.

The model selection step can be avoided by using symbolic regression analysis [Koz92], which claims to find the best model, from the perspective of simplicity and accuracy, that fits a data set. Using such a method requires a larger data set than the one generated in our experiment.

3.2.6 Decision Component

After we described how our framework predicts the performance of an underlying Cloud database cluster, we explain the defining concepts for the decision component.

Now that we have the different cost models as output from the estimation component, the task is to find the values of the independent variables that minimize or maximize the dependent variable. This is a typical combinatorial optimization problem. In other words, the decision component handles the tuning problem as a combinatorial search over a set of system configurations to find a configuration that best achieves the optimization goals of different workloads. Thus, three things must be identified to build the decision component: the search space, the metric for evaluating different options, and the search algorithm.

The metric for evaluating the “goodness” of possible configurations in the search space is measured by calculating the estimated cost and defined based on the optimization goals. Details regarding the search space and algorithm are introduced in the following.

3.2.6.1 Search Space and Candidate Selection

The search space for our tuning problem starts with a space of real continuous values that is defined based on the variable itself. In most cases the value space is changed to discrete positive integers, such as the set of positive integers as a space value for the number of nodes that will be used as a serving infrastructure for the database system. In

the case that the value space is contentious, the search space for the decision algorithm will grow exponentially.

Then an initial step before performing the search algorithm is search space pruning. The value space for each of the variables, which are considered in the configuration, is pruned based on the following:

- Based on infrastructure/physical resources.
- Based on application/workload thresholds.

So, to go on with our example, i.e. the number of nodes for the database system cluster, the value space becomes the integer values between the replication factor and the maximum number of available nodes in the infrastructure or the maximum number of nodes that would be assigned to a workload. If we have an n number of available nodes that we will divide on a set of workloads of size m, where the data replication factor is k , then the number of nodes for a workload will vary between k and n-(m*k) making the search space for the number of nodes [k, n-(m*k)]. The search space is also pruned based on the workloads' performance thresholds, which can be derived from service level agreement or expressed explicitly. An example of such a threshold is a minimum throughput, which is used to prune the search space to start from a minimal number l of nodes that would achieve the specified throughput value and thus service level. The next steps are finding the candidate-solution-space from combinations of the pruned variable sets and then quantitatively comparing different candidate solutions to find an optimal using the search algorithm.

3.2.6.2 Decision Component Algorithm

Algorithm 1: Decision Component Search Algorithm

Data:
WL =$\{w_1, .., w_i\}$ Set of Workloads
w_i =$< wr, r, s, nc, .. >$ Tuple of Workload Characteristics
C =$\{cs, rf, ..\}$ Set of Cluster Configuration Knobs
Result: Pairs of $<wl_i, c>$ that achieve the optimization goal of all WL

1. Perform threshold-based search space pruning
2. Perform infrastructure-based search space pruning
3. Find candidate solutions space S
4. **for** *each candidate solution* **do**
5. | Generate cost function value
6. **end**
7. Perform Combinatorial Search
8. Check objective function value

Given a set of workloads and configurations knobs, the aim of the search algorithm is to efficiently find configuration knobs values that minimizes or maximizes the value

of the goodness metric. For the search algorithm, we are able to use any standard combinatorial search algorithm. In the case of discrete search space, methods such as dynamic programming are well suited. For many contentious search space scenarios, discretizing the search space allows good solutions, for other cases, genetic algorithms can be used to find a solution.

In Algorithm 1, we illustrate the general approach for the decision component without going into detail of the search step 8. Several options exist for performing the search as already mentioned. In Chapter 4 and Chapter 5, we discuss and evaluate the use of brute-force and genetic algorithms.

3.2.7 Monitoring and Refinement Component

After the Cloud storage system deployment, this component starts monitoring and collecting information about the system's performance. In the case that the difference between the model-based predicted value and the monitored value of a performance metric is bigger than a certain threshold defined by the framework user:

$$|Actual_i - Prediction_i| \geq threshold$$

this component should trigger a re-modeling process, which includes:

- Adding the new measurement and removing stale ones.
- Invoking the cost estimation component.

3.3 Framework Usage

After describing the architecture of our framework, we go on explaining the different modes for operating it, based on the ***Self-Tuning Principles*** in Section 2.2.3.1 on page 22, we introduce three modes: Offline, Online, and Offline-Online. These modes differ in the following points:

- Frequency: the modes differ in when and how often the tuning, and modeling processes are performed.
- Scope: the modes differ in what is tuned and if the workload changes are coped with or not.
- Training Data: the modes differ in the training data, which is used for the modeling process.

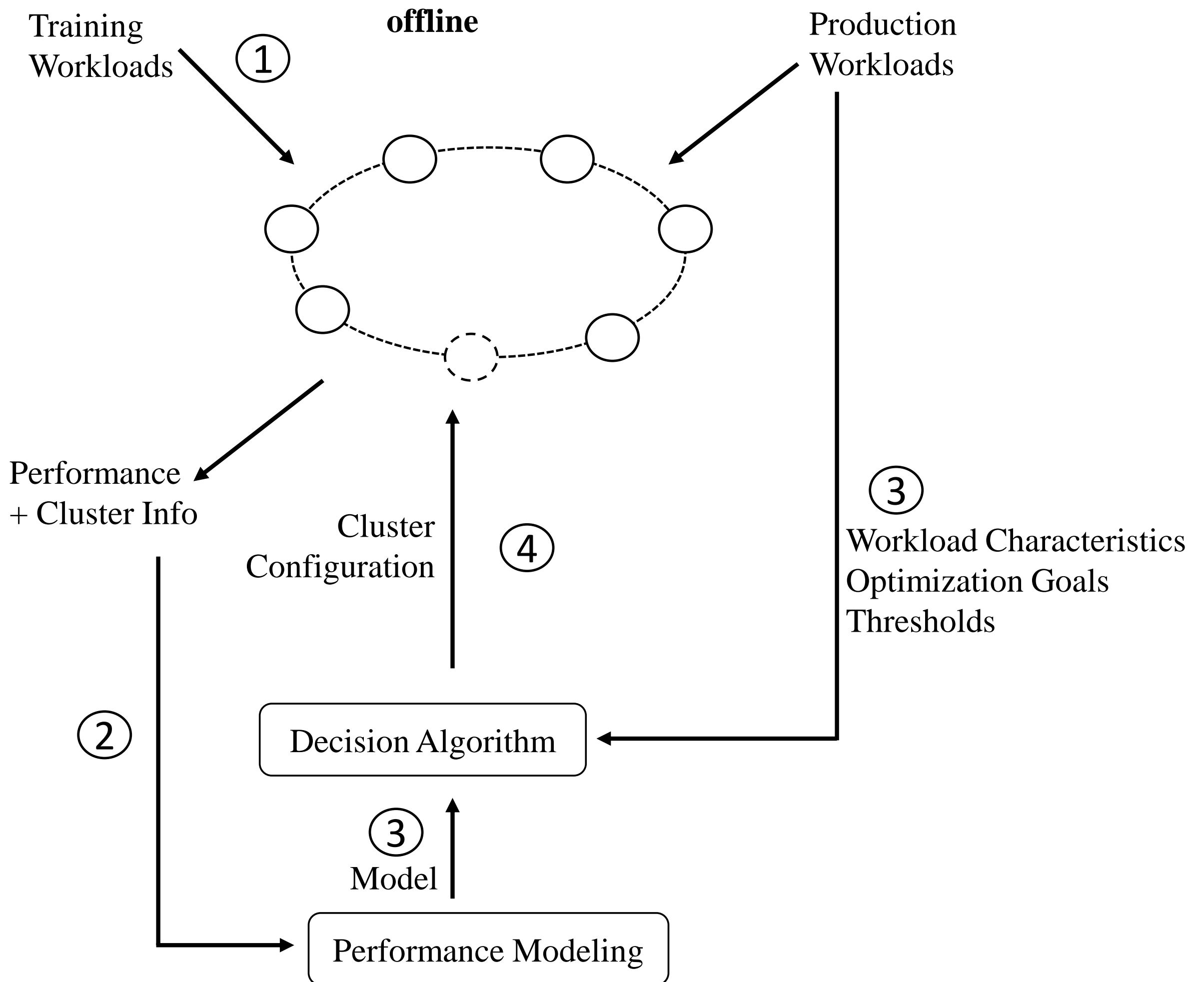


Figure 3.8: Offline Mode of the Framework

3.3.1 Static/Offline Tuning during System Cluster Design

Our first proposal is to run the modeling process when the system is offline during the design phase (i.e. before the system is available to application). Offline performance modeling allows informed configuration decisions and leads to better resource provisioning. The monitoring/refinement component, in this mode, is disabled.

The tuning process, in this case, starts by generating training workloads. We assume that information about the targeted application scenario are available to the framework user. It is then up to her/him to specify training workloads, which resemble the targeted application scenario. The training component deploys the database cluster, generates the workloads and collects information about the performance.

After the training phase, gathered data is analyzed in the cost estimation component to generate a predictive cost/performance estimation model. This model can be used to predict the performance of the database cluster for the targeted workload/workloads with different configurations. Next phase takes place in the decision component.

Using information that includes database cluster settings, production workloads' optimization goals and performance thresholds, the decision component performs search space pruning. The final step in this mode is then to find the cluster configuration. The decision component uses model-based performance predictions to evaluate the "goodness" of possible configurations. It deploys a decision algorithms to find a cluster configuration that best achieves the optimization goal while fulfilling the performance thresholds.

The work-flow of the offline mode of the framework is illustrated in Figure 3.8. We use numbers in this figure to represent the sequence of operations.

3.3.2 Dynamic/Online Tuning during System Cluster Deployment

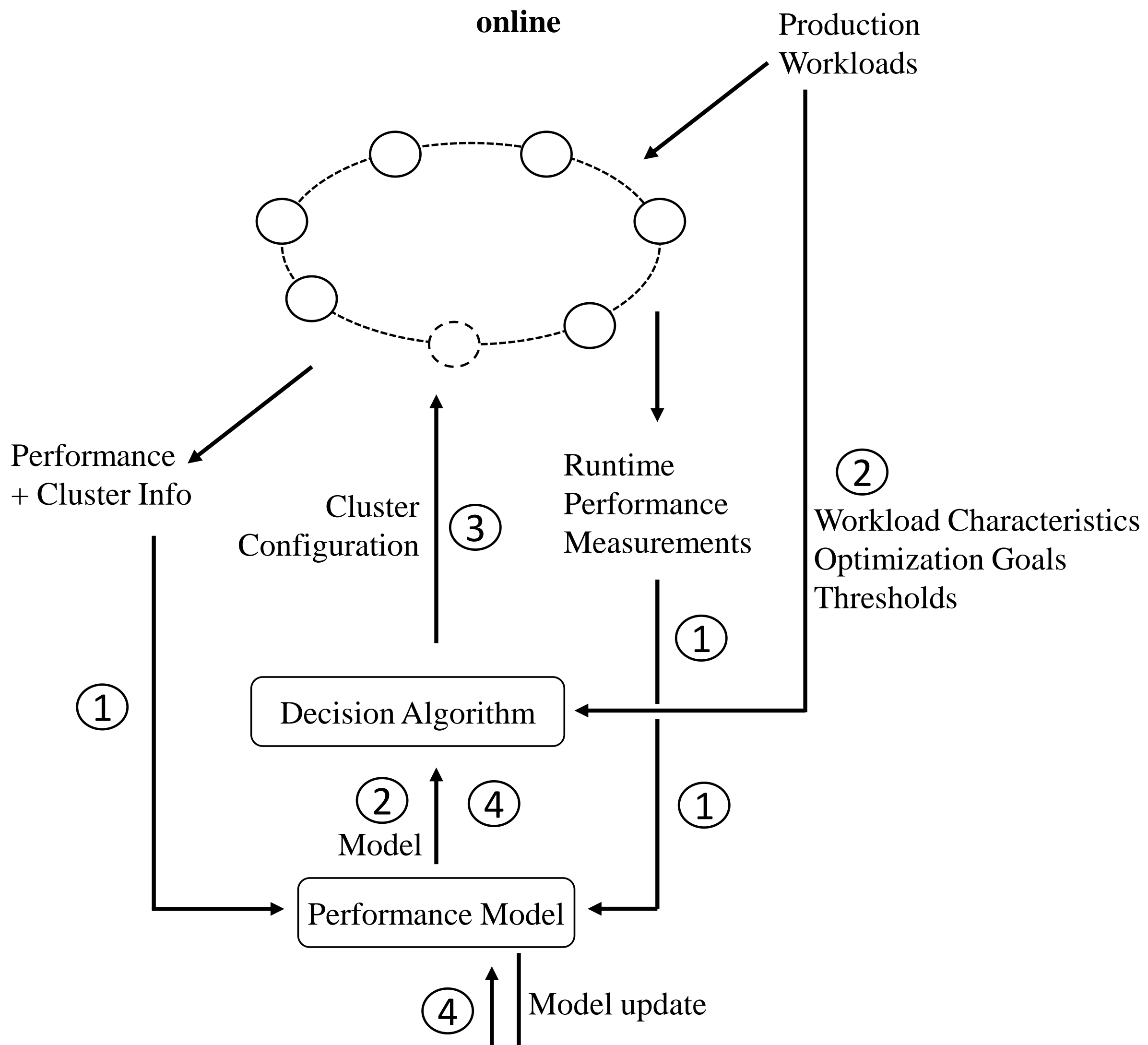


Figure 3.9: Online Mode of the Framework

The second mode for running the framework is the online mode. In this mode, the training component is disabled and no training workloads are run. A model is built only after the database system deployment leaving the initial configuration decision to the system administrator. However, this mode allows the framework to be reactive to changes in workloads and optimization goals and refine the predictive cost estimation models.

The framework monitors the performance of the database system under production workloads and collects statistical data. A change in workload characteristic or goals typically requires changing the cluster configuration. This is initiated by the monitoring/refinement component, which triggers the modeling process once the difference between the workload requirements (i.e. optimization goals and performance thresholds) and the monitored (measured) values is larger than a predefined threshold. An initial prediction model is then built by the cost estimation component. The cost estimation model tests the prediction power of the generated models before using them by the decision component.

The next step is tuning the cluster configuration. Using information that includes the required workload performance, and the database cluster settings, the decision component performs search space pruning. The decision component then uses model-predicted performance values to find the cluster configuration that achieves the optimization goal while fulfilling the performance thresholds of the targeted workloads.

The database system is monitored and information about the performance is collected by the framework. Once the difference between a model-predicted and measured cost values is larger than the predefined threshold, a re-modeling process is triggered and the model is refined.

The online tuning process continues throughout the lifetime of the system. This means that the performance is always monitored and hints regarding the configuration are given.

The work-flow of online mode of the framework is illustrated in Figure 3.9 on the previous page. We use numbers in this figure to represent the sequence of operations.

3.3.3 Offline-Online Tuning Process

Operating the framework, in this mode, allows capturing the benefits of both online and offline modes by enabling the training and the monitoring/refinement components. The framework, in this case, can be used for initial cluster configuration and to react to changes in workloads' characteristics, optimization goals, and performance thresholds.

In this case, the framework starts in the design phase of the database cluster deployment. The training component collects information about the performance of the database cluster with synthetic workloads. As already mentioned, it is up to the framework user to choose training workloads that resemble the targeted application scenario.

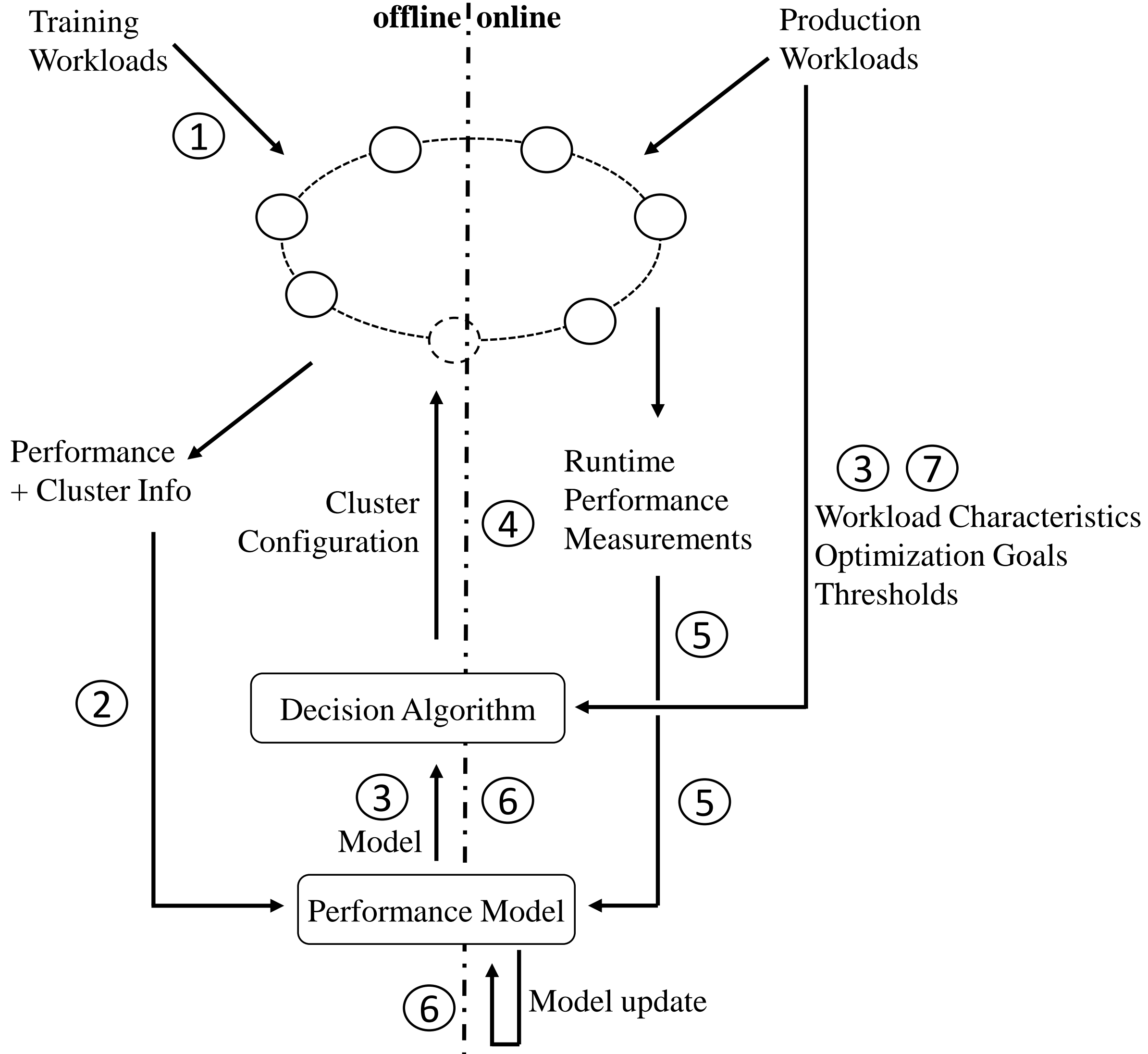


Figure 3.10: Offline-Online Mode of the Framework

After the training phase ends, collected information is analyzed by the cost estimation component in order to create performance models. The models that are generated can then be used by the decision component to advise the database cluster deployment.

The decision component uses the information about the database cluster settings, workloads' optimization goals and performance threshold to perform search space pruning. After that, cluster configuration is found by the decision component's search algorithm. This algorithm depends on model-predicted performance values to validate the possible configurations.

After deployment, the monitoring/refinement component collects information about the performance of the system with production workloads. The collected information is used

to trigger a re-modeling process. This process happens only when the current models fail to provide the required prediction accuracy. This is evaluated by the framework based on predefined thresholds.

The re-modeling process takes place in the cost estimation component. This component analyzes the information containing runtime performance measurements. Using machine learning techniques, it updates and evaluates the cost estimation models. Updating the models during the life time of the system and based on production workloads allows more accurate predictions.

The tuning process continues during the life time of the database system. A change in workloads' characteristics, optimization goals, or performance thresholds typically requires a change in the configuration. Once the difference between the targeted performance metric and its measured value is larger than a predefined threshold, the decision component deploys its search algorithm to find the optimal cluster configuration. The framework user can initiate this process beforehand, when workload change is planned or expected.

The work-flow of the framework, in this mode, is as illustrated in Figure 3.10 on the preceding page. Numbers in this figure represent the sequence of operations.

3.4 Summary

Cloud storage systems' multi-layered modular architecture adds complexity to the tuning process. Moreover, when we have several workloads with different optimization goals being served by one Cloud data storage cluster, the question is how to tune this cluster to best achieve the optimization goals of all workloads. Within this scenario, we defined the tuning problem as:

To find a Cloud storage system cluster configuration c out of a set of possible configurations CC that minimizes (assuming a standard form of the problem) the aggregated costs for all workloads w of a set of workloads WL that need to be supported by the overall cluster while still fulfilling their performance thresholds.

In order to solve this problem, we presented a (self-)tuning framework. We defined the input of this framework as workload characteristics, infrastructure description, and optimization goals. The output was defined as pairs of workloads and cluster configurations. From the architecture point of view, the framework design consists of: training component, cost estimation component, decision component, monitoring/refinement component, and a knowledge-base.

The purpose of the training component is to generate statistical data about the performance of the system under test with different workloads and different cluster configurations. This data is then analyzed by the cost estimation component to generate a performance prediction model. The decision component uses the model to search for the best cluster configuration out of the set of possible configuration. The "goodness" of a candidate solution is measured by the model-based estimated performance value.

After the storage system deployment, the monitoring/refinement component collects information about the system performance. We define the model refinement process to be triggered when the difference between observed and predicted performance metrics is larger than a user-defined threshold.

Additionally, we presented different modes for operating the framework based on the frequency and the scope of the tuning process, and the source of the training data. In the next chapter, we present our implementation of the framework for the offline mode.

4. Prototype Implementation

This chapter shares material with CLOSER13 paper "Clustering the Cloud: A Model for (Self-)Tuning of Cloud Data Management Systems" [MSB13] and the ADBIS15 paper "A Self-Tuning Framework for Cloud Storage Clusters" [MSS15]

In this chapter, we report on our implementation of the self-tuning framework, in the offline mode Section 3.3.1. The purpose of this chapter is to provide a overview for people interested in our approach and discuss the alternatives, which we went through to reach the current implementation.

In our implementation, we focus on modeling the performance characterized by the latency of the Cloud storage system. Latency is modeled in relation to two characteristics: the read/write ratio, and the number of nodes in the underlying cluster, i.e., the cluster size. Later on, the decision component finds the optimal nodes allocation for a set of input workloads while minimizing the sum of latency values for all workloads.

4.1 Required Technologies

In this section, we provide an overview of the technologies and systems, which we used in order to implement the platform and later, conduct the experimental evaluation. These technologies are divided in two parts. The first part considers the programming-related requirements. The second one considers the required underlying infrastructure, which includes software (Cloud storage system) and hardware (cluster of nodes) parts.

4.1.1 Implementation Environment

The prototype for our framework is implemented using Java version 1.7.0_67 for its popularity and cross-platform portability. Most Cloud storage systems provide one or

more Java-based APIs. We exploit its native support of multi-threading for simulating concurrent users [Bre10]. As means of forwarding control commands from the training component manager to the serving nodes, we exploit OpenVPN[1] and Secure Shell (SSH) to forward commands, such as booting/rebooting the storage system and/or operating system from the training component machine (manager) to the storage cluster machines (serving nodes).

In the following section, we provide a short overview of Cassandra's [LM10] basic properties, which are necessary to understand the implementation details of the storage system deployment introduced later in Section 4.2.1.

4.1.2 Cloud Storage System

We designed our framework to work with any Cloud storage system that operates on a cluster of nodes. Examples of such system, are Hive [TSJ+09], Cassandra [LM10], HBase [The15f], Bigtable [CDG+08]. For our implementation, we chose Cassandra as an example of the mentioned systems based of several reasons. On the one hand, it is open source and has the typical properties of NoSQL Cloud storage systems of scalability (running on cluster of nodes, and data partitioning), availability (replication), and partition tolerance. On the other hand, it is popular and has a large user community, which allows good support and maintenance of libraries and bug-fixes.

However, though the performance analysis and modeling, in our design, do not depend on specific storage-system architectural or internal details, the training component implementation does. Our implementation can be extended to support the other systems.

Cassandra was designed for internal use by Facebook messaging system and was later adopted by Apache. Large clusters of Cassandra are currently being used by Apple, Netflix, Spotify [KN10], SoundCloud, and eBay[2], etc.

From the architectural point of view, Cassandra is a peer to peer distributed storage system. Cassandra partitions and distributes the data among nodes in the cluster. It uses a ring representation of the cluster as illustrated in Figure 4.1. The ring representation includes nodes, which map to virtual/physical machines and token values, which are hashed values of row keys. Each node in the cluster is assigned a token value that determines the key range and thus the data for which the corresponding node is responsible.

Balancing the cluster is necessary to avoid bottlenecks and is an important step before running the training workloads by our framework. A Cassandra cluster is balanced by dividing the token-range values equally among the nodes. This requires generating token values, assigning them to nodes, and re-balancing the cluster. This task is performed by our framework as part of the training component manager's tasks. If we assume a partitioner that generates token values range between 0 and 127 and a cluster of four

[1] Information about OpenVPN can be found in: https://openvpn.net/

[2] Details of Use-cases for Cassandra are found on http://planetcassandra.org and http://www.datastax.com/customers

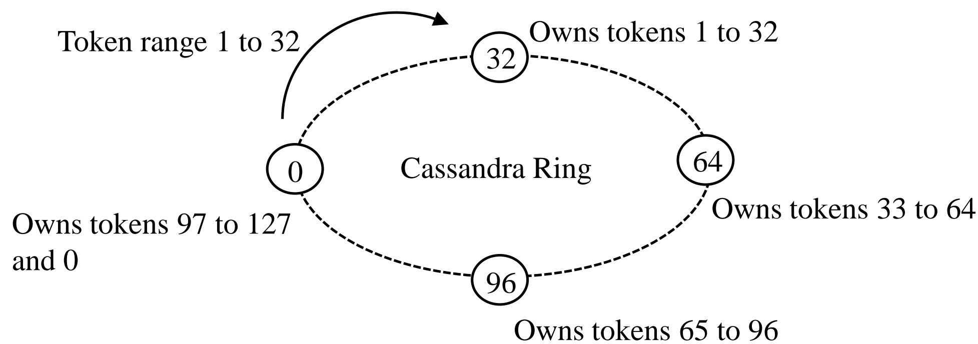


Figure 4.1: Cassandra Ring, adapted from [Nee15]

nodes, the token ownership and distribution among nodes in a balanced cluster can be illustrated as in Figure 4.1.

From the data structure point of view, Cassandra stores data in column families, which can be mapped to tables in the relational world. A column family contains columns of related data where column families that belong to one application are typically organized in one key-space. The schema in Cassandra is flexible, allowing rows with incomplete thus different list of columns as illustrated in Figure 4.2. Cassandra replicates data on the row level. Each replica is stored on a different node and –if applicable– different data center. This is defined by setting the data placement strategy when creating a key-space.

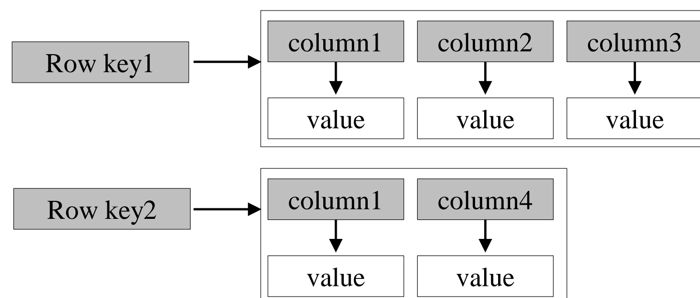


Figure 4.2: Cassandra Column Family

From the data access point of view, Cassandra supports several APIs, a SQL-like query language (CQL)[3], and parallel processing frameworks such as Map/Reduce [DG08], and Spark [ZCD+12]. CQL is typically used for OLTP-like workloads, where as the Map/Reduce framework is used for Online Analytical Processing (OLAP)-like workloads [DG08, ABPA+09]. For the workloads in our framework, we use the Cassandra query language CQL. The modeling and tuning of the parallel processing workloads

[3] http://docs.datastax.com/en/cql/3.0/cql/aboutCQL.html

is already a distinct research area, which has different problems, and scope [PCC+11, JCR11, ZKJ+08].

As means to interact with Cassandra from our framework, we use the DataStax[4] Java library, which is recommended by Apache. Many other Java-based APIs[5] exist, e.g., Hactor[6], Kundera[7], Achilles[8], Astyanax[9], etc., with different support for functionalities, such as connection pooling, load balancing, auto node discovery, connection-retry, etc.

4.2 Implementation

After we introduced the technologies, which we need to build our framework and run the evaluation, we go on to provide an overview of our implementation. As already mentioned in the introduction of this chapter, our implementation is targeted at the offline mode Section 3.3.1 of the framework. In this mode, the framework runs in the design phase of a system which means that the monitoring/refinement component is disabled. The following sections are organized by the consecutive stages of the offline mode and provide implementation details of the involved framework components.

The general process for the framework's offline mode, as illustrated in Figure 3.8, starts by a training phase. A training phase is specified by training workloads settings and based on the replication factor and the maximum number of nodes intended for the database cluster. Several training phases are performed to collect training data. Since we want to model the performance in relation to the the number of nodes, each phase is defined by the number of nodes in the cluster (cluster size). The cluster size is varied between the data replication factor and the maximum number of nodes available. After the training is done for the current number of nodes, it starts again for the next number of nodes. After the training phase ends, data is analyzed to generate the models using an external application. The generated models are used by the decision component to find an optimal allocation of nodes.

4.2.1 Deployment of the Storage Cluster

In this section, we explain the implementation details of the parts responsible for setting up the Cloud storage system cluster and preparing it for the training workloads. This part of the framework is storage system specific, which means that for every Cloud storage system that the framework supports, this part should be extended. The basic classes responsible for setting up a Cloud storage cluster in our framework are illustrated in a Unified Modeling Language (UML) diagram in Figure 4.3 on the next page.

[4]Available for download with other drivers at: https://wiki.apache.org/cassandra/ClientOptions

[5]A list of Cassandra client Java drivers is maintained at: http://www.planetcassandra.org/client-drivers-tools/#Java

[6]Hector available at: http://hector-client.github.io/hector/build/html/index.html#

[7]Kundera available at: https://github.com/impetus-opensource/Kundera/wiki/

[8]Achilles available at: http://doanduyhai.github.io/Achilles/

[9]Astyanax available at: https://github.com/Netflix/astyanax/wiki

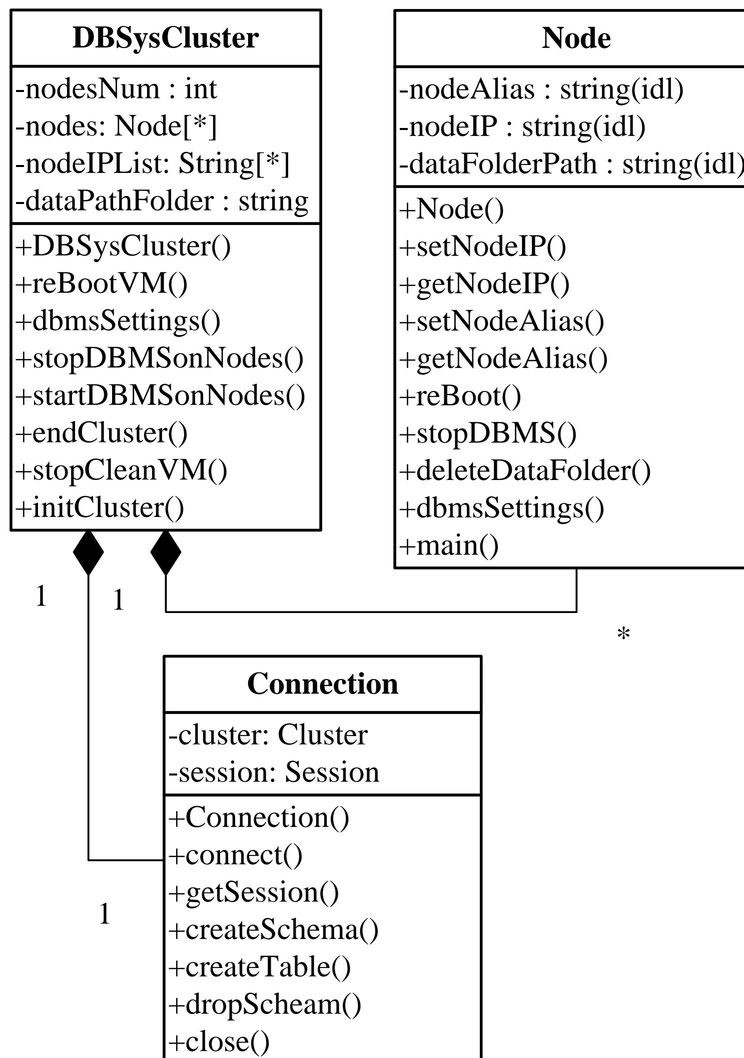


Figure 4.3: Core Classes for the Training Component's Storage System Deployment

Information about the underlying infrastructure that is needed for setting up the storage cluster are extracted from the settings file (e.g., Listing A.2). The cluster size must be changed by the framework for each training phase. We had two choices to implement that:

- Using the Cloud storage system utility of scaling up or down.
- Shutting down the current cluster and starting a new cluster with the new number of nodes.

Neither of these choices changes the evaluation results. However, since the system needs time to stabilize after adding or removing nodes [CST+10], the choice between the two was made in favor of starting a new cluster. The operation of cleaning the cluster for the next iteration of the training phase is explained later in Section 4.2.1.3.

In the following section, we go into important details of Cassandra cluster deployment with predefined configurations as part of the training phase.

4.2.1.1 Cassandra Cluster Configuration

Figure 4.4 illustrates the sequence diagram of the training component manager. The method *initCluster()* is responsible for starting the storage system cluster on the nodes with predefined configuration. This method is database system specific and should be implemented for each storage system when the framework is extended. In our *initCluster()* implementation for Cassandra, we perform a *yaml* file (excerpt illustrated in Listing 4.1) configuration, which includes setting the values for the following parameters:

- ***cluster_name***: Determines the logical cluster that the nodes belong to. It is mainly used to ensure that every machine knows its cluster and prevents it from joining other clusters.
- ***initial_token***: Determines the range of token values that the nodes control and thus determines the data, for which a node is responsible. In the case that virtual nodes approach is used, this value is not set.
- ***num_tokens***: Defines the number of tokens randomly assigned to nodes in the cluster. Assuming the nodes are homogenous and have the same physical resources, they should have the same token values (typically 256). In the case that virtual nodes approach is used, this value is enabled.
- ***partitioner***: Determines how rows are distributed across the nodes in the cluster. The one we used is: org.apache.cassandra.dht.Murmur3 partitioner, which is recommended by Cassandra.

```
# Cassandra storage config YAML
# NOTE:
# See http://wiki.apache.org/cassandra/StorageConfiguration for
# full explanations of configuration directives
# /NOTE
# The name of the cluster. This is mainly used to prevent machines
   in
# one logical cluster from joining another.
cluster_name: 'ClusterOne'
.
.
initial_token: 4192441834933989000
.
# See http://wiki.apache.org/cassandra/Operations for more on
# partitioners and token selection.
partitioner: org.apache.cassandra.dht.Murmur3Partitioner

# Directories where Cassandra should store data on disk. Cassandra
# will spread data evenly across them, subject to the granularity
   of
# the configured compaction strategy.
data_file_directories:
    - /home/student3/cassandra/data

# commit log
commitlog_directory: /home/student3/cassandra/commitlog
.
.
seed_provider:
    # Addresses of hosts that are deemed contact points.
    # Cassandra nodes use this list of hosts to find each other
       and learn
    # the topology of the ring. You must change this if you are
       running
    # multiple nodes!
    - class_name: org.apache.cassandra.locator.SimpleSeedProvider
      parameters:
          # seeds is actually a comma-delimited list of addresses.
          # Ex: "<ip1>,<ip2>,<ip3>"
          - seeds: "192.168.144.11,192.168.144.12,192.168.144.13"
.
.
```

Listing 4.1: Excerpt of the Cassandra Configuration File

- ***data_file_directories***: Determines the directory path where tables' data will be stored.
- ***commitlog_directory***: Determines the directory path where log data will be stored. It is preferable to have it on different disk partition or different physical device –when applicable– from where table data is stored.
- ***seeds***: IP address list of other nodes in the cluster. This list is used by the gossip protocol to bootstrap a newly added node. It should include the IP address of a node from each data center if applicable. It is also recommended to have more than one node from each data center for fault tolerance.
- ***listen_address***, ***rpc_address***: Both are set to the value of the IP address of the corresponding node. The first one determines the address or host name that other nodes in the ring can use to communicate with this node. The later determines the listen address for client requests.
- ***rpc_port***: Determines the port for data access client connections. Typical value for this port is set to 9160.
- ***endpoint_snitch***: Determines the network topology, which Cassandra uses for request routing and replica distribution. Cassandra supports multiple topologies. The one that is suitable for our implementation, since we have only one data center, is SimpleSnitch[10].

After the *yaml* file is configured correctly on all the nodes that should belong to the cluster, Cassandra is started using the ssh command that is triggered from the training component manager to the serving nodes. The next step of the process would be creating the database schema that will be used by the training workloads. In the coming section, we illustrate general activities involved in this step.

4.2.1.2 Schema and Data Generation

The schema is provided by the framework user in the XML configuration file Listing A.2. For Cassandra, the schema definition includes key-space, column families (tables), replication factor, and the data placement strategy. Depending on what the framework user needs, the framework allows table definition as:

- Number of columns of string values.
- Complete schema definition.

[10]Cassandra SimpleSnitch: https://docs.datastax.com/en/cassandra/1.2/cassandra/architecture/architectureSnitchesAbout_c.html

For simple and general workloads, the training step deploys tables defined by number of columns of string values. The Listing 4.2 illustrates the minimal code for creating key-space and column family in the *initCluster()* method. In this example, which we used for creating the column families for our evaluation, a column family having $columnNum$ as the number of columns and text as data type, is created.

```
1  Session session= connect(nodeIp);
2  session.execute("CREATE KEYSPACE "+ keyspace +" WITH replication "+"= {'
       class':'"+strategy+"', 'replication\_factor':"
3                      +replicationFactor+"};");
4                      String query="CREATE TABLE "+keyspace+"."+tableName+
                          " ("+"id int PRIMARY KEY"   ;
5              for (int i = 1; i < columnNum; i++) {
6                  query=query+", "+"column"+i+" text";
7              }
8              query=query+ ");";
9              session.execute(query);
```

Listing 4.2: Minimal Java Code for Creating Cassandra Key-space and Column Family

For a specific training workload, the framework user can set detailed schema. Example of this is illustrated in Listing 4.3.

```
<Database DPStrategy="SimpleStrategy" keySpace="sibaKS"
   replicationFactor="3">
       <table tName="users">
              <column name="id" dataType="uuid"/>
              <column name="user_name" dataType="ascii"/>
              <column name="password" dataType="ascii"/>
              <column name="email" dataType="ascii"/>
              <column name="gender" dataType="text"/>
              <column name="phoneNum" dataType="text"/>
              <column name="country" dataType="text"/>
              <column name="birthY" dataType="bigint"/>
       </table>
</Database>
```

Listing 4.3: Excerpt of the Training Component Settings: Defining Database

The schema is created at the end of the cluster initiation phase as shown in the training component work-flow in Figure 4.4 on the next page. The *initCluster()* method performs key-space and column family creation by passing CQL statements using the Datastax Java-API to the Cassandra cluster.

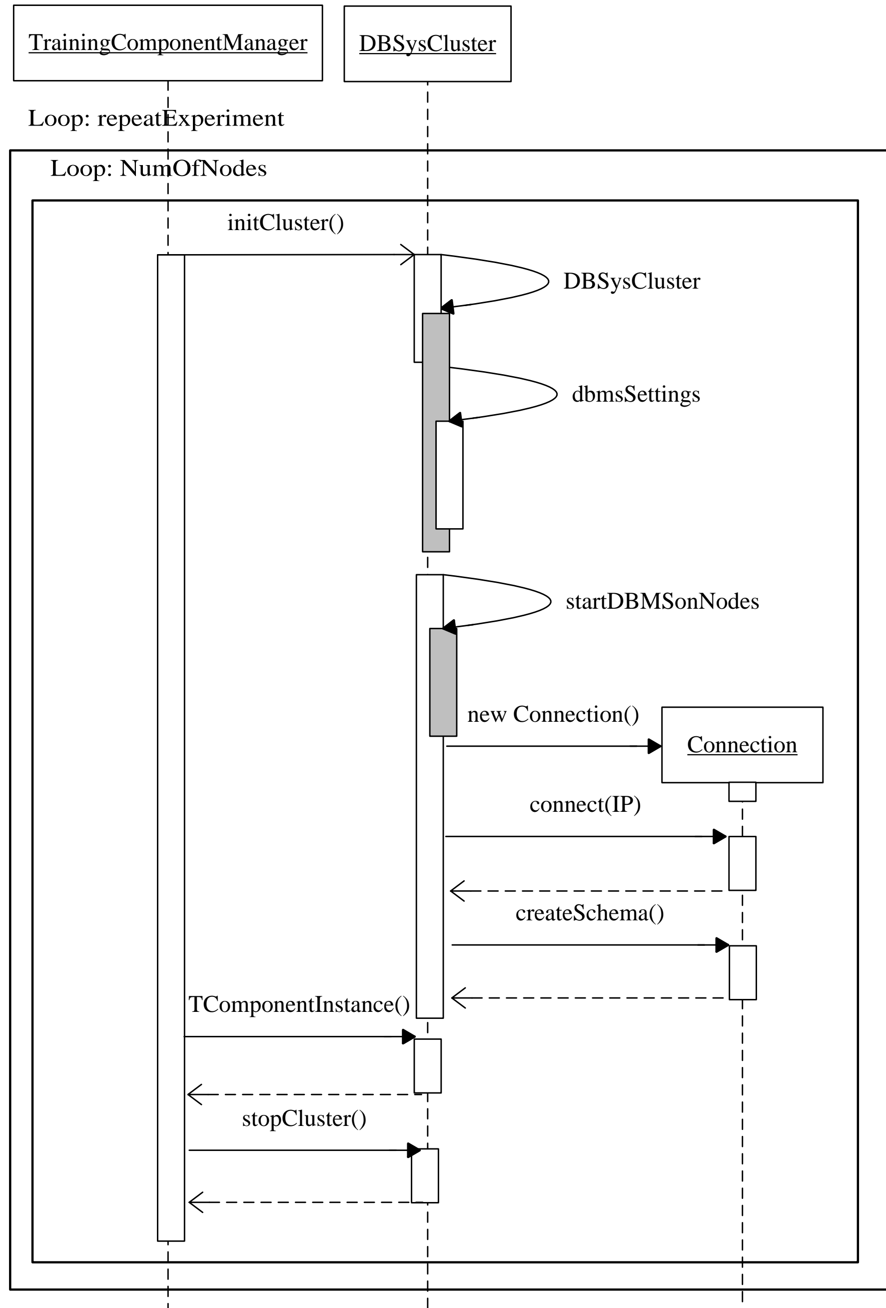


Figure 4.4: Sequence Diagram of the TrainingComponentManager's Main Loop

4.2.1.3 Cleaning the Cluster

One of the important tasks of the training component is to clean the infrastructure and system for the next iteration of the training phase. In our implementation, this is achieved through the method *stopCluster()*. This method involves three steps:

- Shutting down the storage system on every node involved in the cluster. This method is storage system specific and must be adapted for other storage systems when the framework is extended.
- Deleting the data folders on the all nodes.
- Rebooting the nodes (virtual machines in our case) to clear any caches that could affect the performance.

The sequence diagram for the *stopCleanVMSeq()* is presented in Figure 4.5 on the following page.

4.2.2 Workload Generation

In this section, we provide an overview of the implementation of the parts responsible for generating the workloads for the training phase. The core classes responsible for workload generation are illustrated in Figure 4.6 on page 76.

Based on the application scenario for which the tuning is planned, the user can define a training workload. Though in the relational database systems' world, there is an enormous number of performance benchmarks that define tasks and data sets derived from and representative of real use case scenarios [Raa93, Ser91, Nel91]. To the best of our knowledge such a standardization for Cloud storage systems does not exist until now. Multiple performance benchmarks use synthetic workloads, such as the widely used Yahoo! benchmark [CST+10] and the Bigtable benchmark [CDG+08]. Thus, our training component does not include out of the box training workloads that map to a real scenario, rather allows the framework user to define them based on the expected application. Workloads are defined by the framework user in the XML framework settings file Listing A.2 and are designed to be a mix of the following operations:

- Sequential write (bulk write), such as in backup and extract. Writes rows into a table (column family) under sequential row keys.
- Sequential read (scan), such as in import operations. Reads rows under sequential row keys.
- Random write (point write), such as in batch update. Writes rows into an table (column family) under random row keys.
- Random read (point selection, i.e., one row is returned for each call to the DB). Reads rows under random row keys

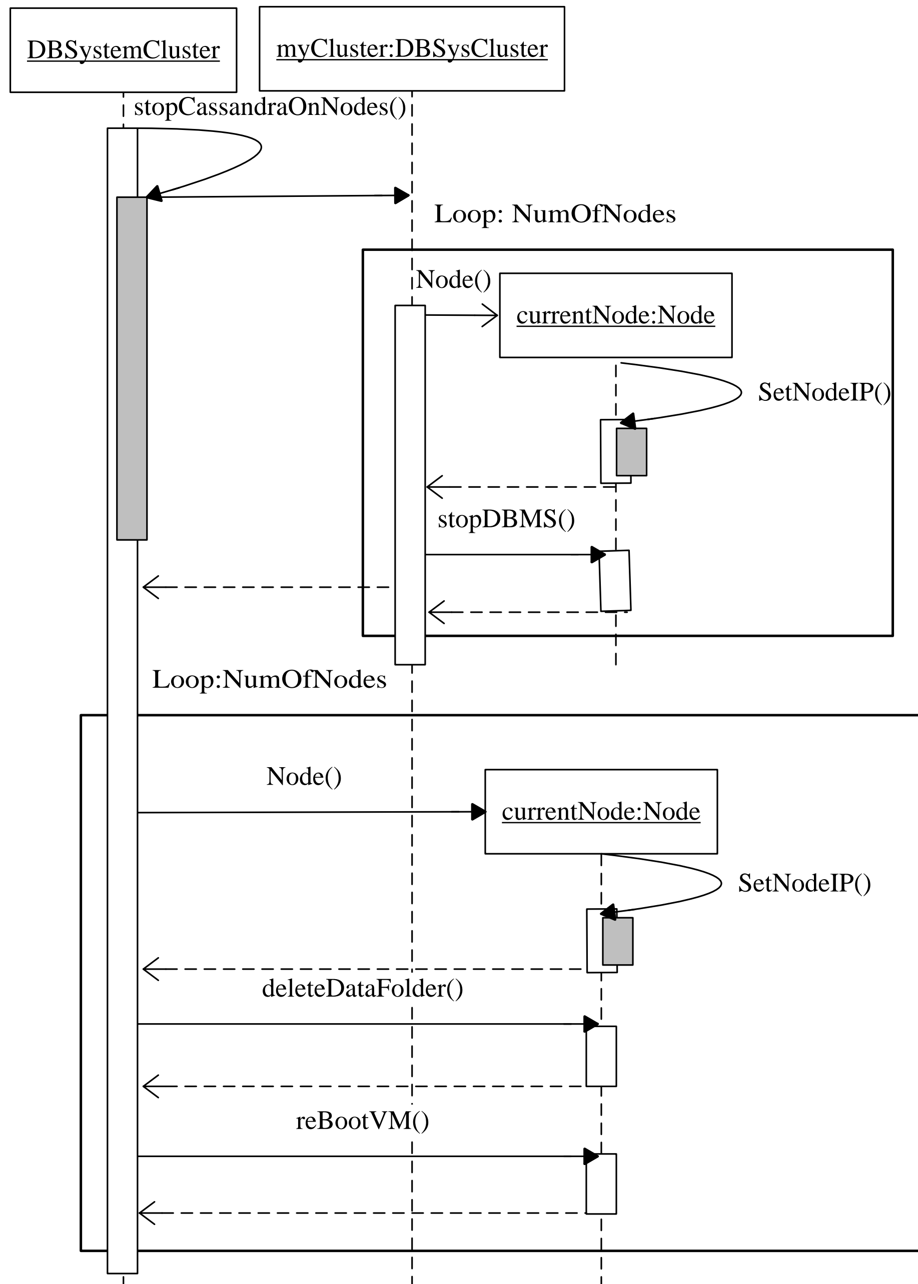


Figure 4.5: Sequence Diagram of the stopCluster Method

The implementation can be extended to include other operations, such as update, read latest, range selection. The workload can have different percentages of operations in the mix, different request number, different parallelism degrees. Besides configuring the database schema and queries, the training workloads are characterized by operation throughput. For that, we allow setting the number of the training components instance (which would start a new Java client for each instance on the training component instances machines) and the number of concurrent threads (each training component instance would start a number of concurrent threads). An example of workload definition is illustrated in Listing 4.4.

```
<TrainingWorkload repeat=”20” seqWritePercentage=”100”
   requestNum=”520000”
   measurementFolderPath=”/home/training/measurement”/>
<TISettings>
      <node IPAdress=”192.168.144.24” clientNum=”3”
         threadsPerClient=”5”/>
      <node IPAdress=”192.168.144.25” clientNum=”3”
         threadsPerClient=”5”/>
      <node IPAdress=”192.168.144.26” clientNum=”3”
         threadsPerClient=”5”/>
</TISettings>
```

Listing 4.4: Excerpt of the Training Component Settings: Defining Workload

The minimal code, for creating the CQL statement for write queries, is illustrated in Listing 4.5. We use *SequenceGenerator* to generate sequential numbers to be used a key value. We also use *RandomStringGenerator* to create random string values for the insert operation. Both generators are defined in the training component.

In our implementation, the performance is characterized by measuring the time required to finish the whole workload (a number of requests with a level of parallelization) and not for single operations. For that the time is measured before and after the workload. Each training workload is repeated several times and the average latency is calculated. The sequence diagram for starting a training workload is illustrated in Figure 4.7.

4.2.3 Data Analysis and Modeling

After the training phase, the cost estimation component starts the process of model learning from the training data as defined in Section 3.2.5. The first step is preparing the data for the learning process. This involves extracting and aggregating the relevant data from the training measurement files. The metric, that we consider through out the thesis, is the average latency value. For implementing the model learning step, we consider the following alternatives:

- Regression analysis [APB15].

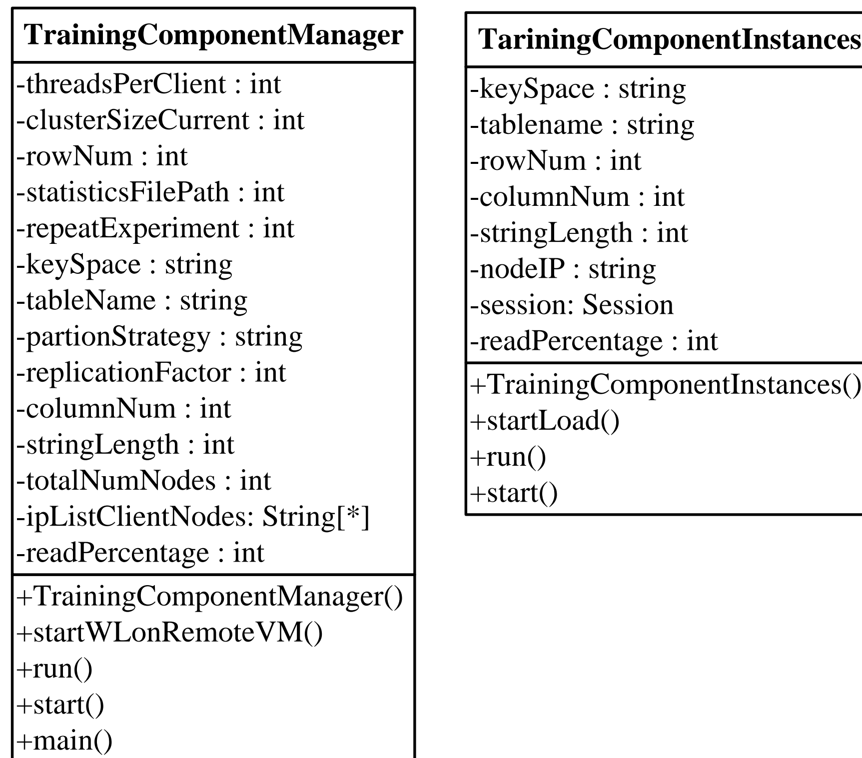


Figure 4.6: Core Classes for the Training Component's Workload Generation

- Symbolic regression [KSV08].

The advantage of using the regression analysis is that it is simple and straight forward with little or no requirements for pre-configuration. The other option is using symbolic regression technique, which by using genetic programming considers and evaluates larger number of models compared to the first option. It allows model-complexity control by allowing the configuration of the model's mathematical operations. The disadvantage of using symbolic regression is its higher overhead. The search space can be explicitly restricted by limiting the mathematical operations that can be used in generating the model and by explicitly setting the genetic algorithm to stop after a specific time. This leads us to the second disadvantage of the symbolic regression technique: its complexity regarding the large number of parameters that should be pre-configured and have a direct effect on the form and validity of the generated model.

Both considered alternatives for the learning step have the following implementation options:

- Libraries implemented in Java, such as the Apache Commons Mathematics library[11].

[11] Apache mathematics library available at: https://commons.apache.org/proper/commons-math/

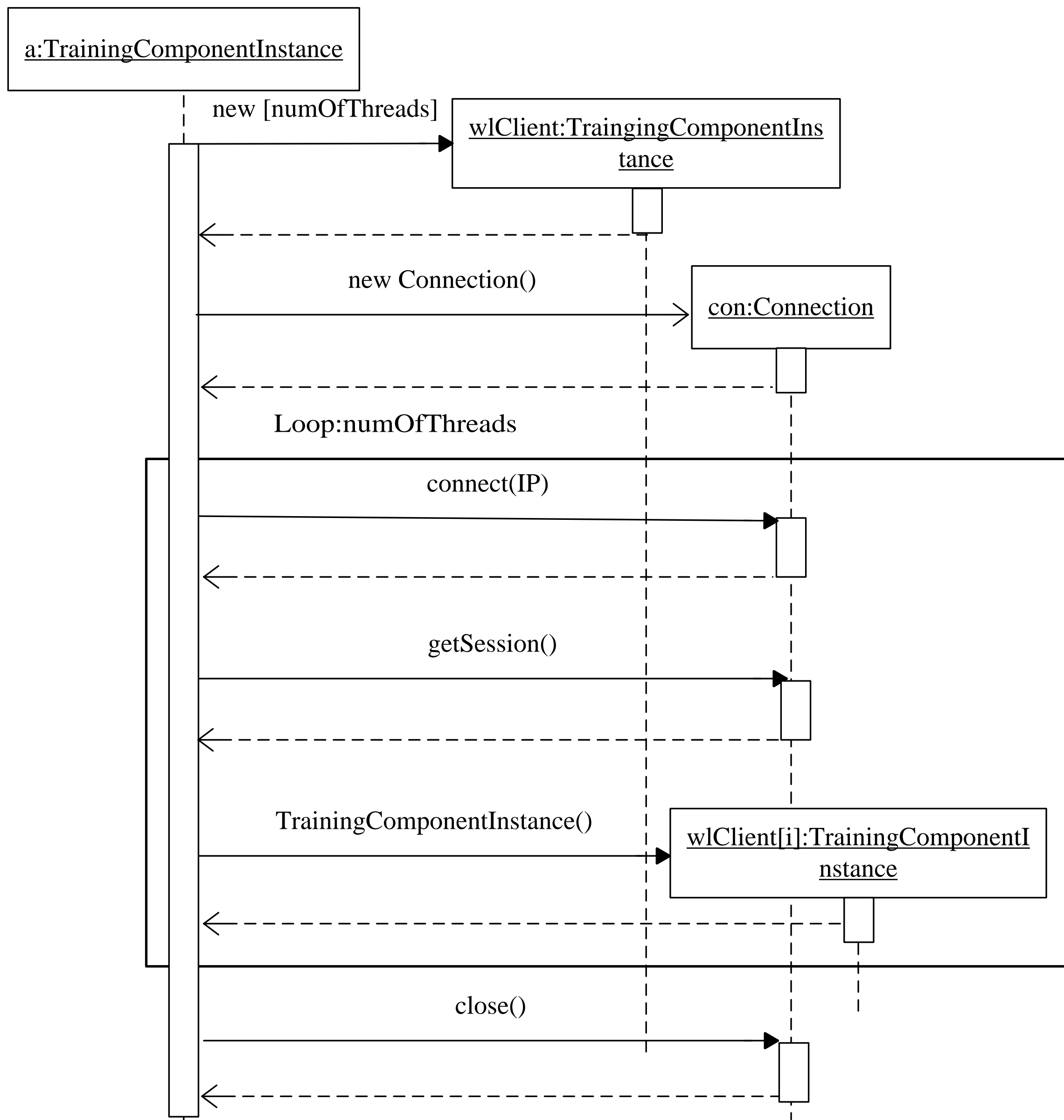


Figure 4.7: Sequence Diagram of the startLoad Method

```
1       SequenceGenerator sequence=new SequenceGenerator();
2       for (int i = 1; i < rowNum; i++) {
3           //create id value
4           long idValue=sequence.getnext();
5           String query="INSERT INTO "+keyspace+"."+tableName+" (id";
6           for (int j = 1; j < columnNum; j++) {
7               query=query+", "+"column"+j;
8           }
9           query=query+") VALUES ("+idValue;
10          for (int j = 1; j < columnNum; j++) {
11                  query=query+",'"+RandomStringGenerator.
                        generateRandomString(stringLength,
                        RandomStringGenerator.Mode.ALPHANUMERIC)+"'"}
12          } //for end
13          querym=query+ ");";
14          session.execute(querym);
```

Listing 4.5: Minimal Java Code for Write Operation

- External tools, such as Matlab[12], Maple[13], and R[14].

In the first option, there are several statistical mathematical libraries with different ranges of supported regression techniques. Using this option allows centralized control of framework functionalities and redundancy free representation of models and data. However it provides less options than external tools.

The use of an external tool enables accessibility to a wide range of machine learning techniques. It is also possible to perform the prediction step in the external tool and then retrieve the result to the framework. To integrate an external tool with our framework, the following steps are performed:

- Initiating an external tool engine for the current session.
- Preparing and exporting data.
- Calling data analysis methods of the external tool and retrieving the result.

For our implementation, we deploy regression analysis techniques in an external tool. Many tools provide libraries to connect to Java such as Maple OpenMaple library[15] and R rJava library[16]. We choose R since it is open source, popular, and supports a wide variety of machine learning techniques. An excerpt example code for operating R from Java is illustrated in Listing 4.6.

[12] Matlab available at: http://de.mathworks.com/products/matlab/
[13] Maple available at: http://www.maplesoft.com/compare/mathematica_analysis/
[14] R available at: https://www.r-project.org/
[15] OpenMaple library available at: http://www.maplesoft.com/support/help/Maple/view.aspx?path=OpenMaple/Java/API
[16] rJava available at: https://cran.r-project.org/web/packages/rJava/index.html

```
1  public class R {
2  public static JRIEngine rEngine;
3  public R() throws REngineException{
4          // Start R session
5                  Rengine re = new Rengine (new String [] {"--vanilla"},
                       false, null);
6                  // Check if the session is working.
7                  if (!re.waitForR()) {
8                      return;
9                  }
10         this.rEngine=new JRIEngine(re);
11     }
12
13     public JRIEngine getR(){
14         return this.rEngine;
15     }
16
17 public static void main(String[] args) throws REngineException,
       REXPMismatchException {
18 R r = new R();
19 Rengine rEngine = r.getR();
20 //Preparing and Inserting Data to R
21 rEngine.assign("Latency", prepareLatencyAvgArray(trainingDataFile);
22 rEngine.assign("writeRead", preparewRArray(trainingDataFile));
23 rEngine.assign("clusterSize", prepareClusterArray(trainingDataFile));
24 //Preparing Dataframe
25 String cmde=
26     R.makeRstmtDataframe("df","writeRead","clusterSize","Latency");
27 rEngine.parseAndEval(cmde);
28 //Performing Regression
29 REXP result=
30     r.rEngine.parseAndEval(lm(Latency ~ writeRead + clusterSize),df);
```

Listing 4.6: Excerpt Java Code for Using R with Java in our Framework

Other than the mentioned advantages and disadvantages, we will discuss, in the next chapter, the validity of this option and its limitations with regards to our use case.

The generated performance models are used by the decision component to find the optimal cluster configuration that achieves the workloads's optimization goals and fulfills their performance thresholds. In the following section, we provide details about the search algorithms deployed in our decision component.

4.2.4 Decision and Search Algorithm

As a basic step for the tuning process, we focus on finding the optimal allocation of nodes for each workload while minimizing the sum of latency values for all workloads. In order to achieve that, we modeled the latency in relation to workload characteristic: write/read ratio and cluster configuration: number of nodes.

Algorithm 2: The Search Approach using Brute-Force

Data: $WL = \{w_1, .., w_i\}$ Set of Workloads
$w_i =< wr, r>$ Tuple of Workload Characteristics
$C = \{cs\}$ Set of Cluster Configuration Knobs
Result: Pairs of $<wl_i, n_i>$ that achieve minimum latency

```
1  Perform threshold-based search space pruning
2  Perform infrastructure-based search space pruning
3  Find candidate solutions space S
4  for each candidate in S do
5      for each workload in candidate do
6          Calculate latency value based on the model
7      end
8      Assign minimum latency
9      Assign sum of latency as candidate latency
10     if candidate latency<minimum then
11         Assign Candidate as solution
12         Assign new minimum latency
13     else
14         if candidate latency=minimum then
15             Choose one with minimal number of nodes
16             Assign Candidate as solution
17         end
18     end
19 end
```

Based on that, the concrete problem that our implementation solves, is a specialization of the problem defined in Section 3.2.1 on page 41, and is defined as follows:

For each workload $w \in WL$, find the number of nodes n_w that achieves:

$$\sum_{w \in WL} latency(w, n_w) \rightarrow min$$

subject to

$$\sum_{w \in WL} n_w \in \left[\sum_{w \in WL} k_w, .., N \right]$$

where k_w is the data replication factor for workload w, and N is the total number of nodes in the infrastructure.

The input for the decision component is workloads' characteristics represented by read-/write ratio, cluster configuration represented by its size, and the model generated from data analysis and modeling step.

To perform the search for an optimal solution, we consider two options:

- Brute-force [HDBD09].
- Genetic algorithms [Mit96, HDBD09].

Based on which, we modify our initial Algorithm 1, presented in Section 3.2.6. The outline for both algorithms is illustrated in Algorithm 2 on the preceding page and Algorithm 3 on the following page, in consecutive order.

These two algorithms are a specialization (regarding optimization goal) and an extension (regarding steps 4,..,8) of Algorithm 1. Compared to Algorithm 1, the cost function is represented here by the latency value of the storage system. Consequently, the objective function is calculated by the sum of latency for all workloads and should be minimized.

	Brute-force Approach	Genetic Algorithm Approach
Deterministic	yes	no
Overhead	proportional to search space	configurable
Solution	optimal	depends on num of generations

Table 4.1: Comparison of Algorithms Considered for the Decision Component

In the ***brute-force*** based approach, Algorithm 2, the first step is search space pruning. In our implementation, the search space pruning is based on available infrastructure and workloads' performance thresholds as already defined in Section 3.2.6.1. The search space for the number of nodes starts by the sum of data replication factor of all workloads and ends with the maximum number of nodes available in the infrastructure.

After the search space is defined, the brute force algorithm iterates over model-based predicted values of latency to identify the number of nodes that would achieve the minimal sum of latency for all workloads. In the case of multiple options achieving the minimal latency, the solution with minimum sum of assigned nodes is given advantage. The brute-force algorithm is easy to implement and will always find the solution for this use case. However, its cost is proportional to the number of candidate solutions.

In the ***genetic algorithm*** based approach, Algorithm 3, the framework starts by an initial candidate solution, which is generated by an even distribution of nodes onto

workloads. For a specified number of generations, mutations on the initial candidate are generated by random increase/decrease of the number of nodes assigned to each workload. For each candidate solution in the candidates' pool of a generation, the nodes number should adhere to the conditions of individual workload's replication factor and the global sum that should be less than N.

The fitness of a candidate is calculated based on model-predicted latency values. Just like the previous approach, a candidate with the minimum sum of nodes is given advantage. The genetic algorithm terminates after a configurable number of generations. It can also be set to stop after a certain amount of time. The overhead of this approach can be configured by the number of generations and the pool size for each generations. This approach does not guarantee an optimal solution. However, its cost is typically small compared to the brute-force approach. Table 4.1 includes comparison of the two approaches. In the next chapter, we provide more insight into this comparison.

Algorithm 3: The Search Approach using Genetic Algorithm

Data: $WL = \{w_1, .., w_i\}$ Set of Workloads
$w_i = < wr, r >$ Tuple of Workload Characteristics
$C = \{cs\}$ Set of Cluster Configuration Knobs
Number of generations, Pool size
Result: Pairs of $< wl_i, n_i >$ that achieve minimum latency

1 Create an initial candidate solution with even distribution of nodes
2 **for** *each workload in candidate* **do**
3   Calculate latency value based on the model
4 **end**
5 Assign sum of latency as candidate-Latency
6 Assign candidate latency as minimum
7 **for** *each generation* **do**
8   Create candidates pool: mutations on nodes number assigned for workloads
9   **for** *each candidate in the pool* **do**
10   Calculate candidate latency
11   **if** *candidate latency<minimum* **then**
12   Assign Candidate as solution
13   Assign new minimum latency
14   **else**
15   **if** *candidate latency=minimum* **then**
16   Choose one with minimal number of nodes
17   Assign Candidate as solution
18   **end**
19   **end**
20   **end**
21 **end**

4.3 Summary

In the previous sections, we reported on the prototype implementation of our framework. First, we provided an overview of technologies and systems, which were used in order to implement the framework and later conduct the evaluation. The framework was implemented in Java, where Cassandra was used as example of the underlying Cloud storage system.

As a basic step for the tuning process, our implementation focused on modeling the performance of the underlying storage system characterized by latency in relation to workloads' and cluster configuration criteria. For that purpose, we implemented the training component to perform several training phases that include different workloads and different cluster configurations. Then we provided details about the steps involved in the training phase, which include: preparing the storage system cluster, creating the database, generating workloads, and collecting performance metrics.

After that, we discussed different alternative for the data analysis and modeling step. For our implementation, we deployed regression analysis techniques in an external tool.

Last, we presented two alternative algorithms for the decision component: the brute-force algorithm and the genetic algorithm. We discussed both algorithms and their differences. In the next chapter, we evaluate the validity of both algorithms based on experimental results.

5. Experimental Setup and Evaluation

This chapter shares material with the ADBIS15 paper "A Self-Tuning Framework for Cloud Storage Clusters" [MSS15]

In this chapter, we present the evaluation of our framework in the offline mode. We start by giving an overview of the experimental setup. This includes the underlying infrastructure, and the settings for the storage system cluster. After that, we introduce the training workloads, training data, and the result from the modeling process. At the end, we discuss alternatives of the search algorithm for the decision component.

5.1 Experimental Setup

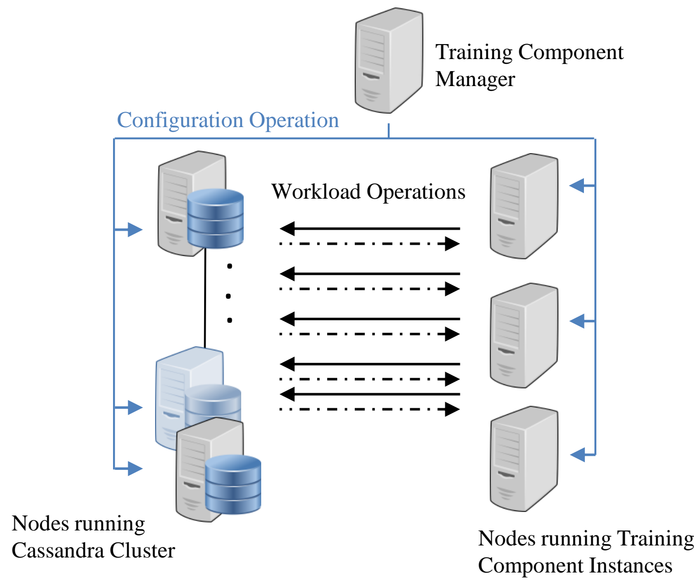
In this section, we introduce the hardware and software configuration of the infrastructure and the storage system, which are used to conduct the experiments.

5.1.1 Infrastructure

First, we provide an overview of the needed infrastructure. To conduct our experiments, we require a network of machines (computers). For that, we create a network of 15 virtual machines using VMware ESXi[1], which is a bare-metal hypervisor. VMware runs directly on the hardware without a host operating system. The settings of the virtual machines and the used infrastructure are listed in 5.1b on the next page and 5.1c on the following page consecutively.

Since the aim is to model the storage system performance, it is best to run the framework on machines different from the ones that are used for the storage system deployment.

[1]VMware available at: https://www.vmware.com/products/vsphere/features.html



(a) Deployment on Network of Virtual Machines

OS Ubuntu: 13.04 kernel	Version: Linux 3.8.0-35-generic-pae
CPU: Intel(R) Xeon(R) E5-2650 2.00GHz 2 Cores	Cache size: 20480 KB
Disk: 90.18 GB 7200RPM	Memory: 8 GB
Network: 100 MBits	Java Version: 1.7.0_25
Cassandra Version 1.2.13	Virtual machines for generating workload: 3
Replication factor: 3	Virtual machines for cluster deployment: 11

(b) Software and Hardware Configuration of the Virtual Machines

CPU: Intel(R) Xeon(R) E5-2650 2.00GHz 8 Cores	Cache size: 20480 KB
Total number of physical machines: 8	Memory: 34 GB
Network: 100 MBits	VMware Version: 5.5.0

(c) Hardware Configuration of the Physical Machines

Figure 5.1: Infrastructure used for the Prototype Deployment

A number of the virtual machines is used for the framework deployment. The training component manager is installed on one machine and it controls the initiation of the training component instances on other machines. The number of machines that are used for the training component instances varies from one workload to the other and depends on the required throughput. The remaining and larger part of the network is used for the deployment of the storage system cluster. The framework deployment on infrastructure is illustrated in 5.1a.

5.1.2 Cassandra Deployment

In our experiment, we use Cassandra version 1.2.13. The framework manager deploys Cassandra automatically before each training phase according to the specified configuration. We installed Cassandra on each of the nodes, which will participate in the training phase. The framework manager starts a configuration operation which includes:

- Manipulating the Cassandra *yaml* file, changing the values of the parameters discussed in Section 4.2.1.1 on all the nodes that will participate in the cluster at the corresponding training phase.
- Start up of a Cassandra instance, using shell script `.\bin\cassandra`.

Other Cassandra settings are left to the default values. At this stage, our goal is not to tune Cassandra performance for a certain workload, but rather model its performance with different parameters and different workloads. Such parameters will be varied and the performance of the Cassandra cluster will be observed then with different workloads to be able to generate a useful model.

5.2 Experiments

The workloads and data, which we used in our evaluation are synthetic. As already mentioned in the implementation chapter, Cloud storage systems developers such as Google and Yahoo! introduced such synthetic workloads to test the performance of their systems [CST+10, CDG+08]. In the following sections, we introduce the results of evaluating different aspects of the framework. The goals of the experimental and evaluation phase are the following:

- Model the performance of the storage cluster.
- Evaluate the alternative regression techniques used for modeling.
- Evaluate the prediction power of the models.
- Evaluate the algorithms presented in Chapter 4: brute-force-based Algorithm 2, and genetic-based Algorithm 3 as the search algorithm for the decision component.
- Evaluate the decision made by the framework.

5.2.1 Experiment Design

With the goal of modeling the performance of the database cluster with different cluster sizes and different workloads, we designed the following experiments:

Varying the Number of Nodes

The number of nodes is the cluster configuration parameter, which will be one of the independent variables in our model. Depending on the maximum number of nodes that we have in our infrastructure 11, and the replication factor 3, the framework varies the number of nodes in the cluster between 3 and 11. Each training phase increases the number of nodes by one. In the case that the available infrastructure allows larger number of nodes in the serving cluster, it is possible to minimize the number of iterations of the training phase by increasing the number of nodes between iterations by more than one.

Varying the Write/Read Percentage

The write/read percentage is the workload characteristic parameter, which will be one of the independent variables in our model. In this case, the framework generates in each training phase a workload with a certain percentage of write/read operations in each. Since we only use two kinds of operations in the workloads, i.e. random read and random writes, the percentages is identified by the write percentage and varies between 0 and 100. However, in our implementation we support sequential read and writes so other workload mixes are possible.

Other workload criteria, such as the schema, consistency level, row size (which determines, together with request numbers, the data volume of the workload), and goal throughput (concurrent accesses) are kept the same. As a performance metric, which will be used as a dependent variable in our case, we monitored the latency of the system. We consider the latency of the system for a whole workload.

Now that we have the outline for the experiments, which we are going to conduct, let us take a look at the details of the generated training workloads and database.

5.2.2 Database

For our tests, we create a key-space with a replication factor of 3 (the typical replication factor). This choice of replication factor leads to a minimum of 3 nodes for cluster size. The other criteria that should be specified when creating the key-space is the placement strategy of data replicas. In our case, since we are working on nodes within one data center, we use the *SimpleStrategy*. If the deployment is going to take place in more than one data center, the *NetworkTopologyStrategy* should be used as the data placement strategy. The following statement is used to create the Cassandra key-space:

```
CREATE KEYSPACE TWkeyspace WITH replication={'class' : 'SimpleStrategy',
'replication_factor' : 3}
```

The training component created a column family for each workload. We defined each column family to be made of 20 columns, one of which is a key. The remaining columns are defined as string values. The following statement was used to define the column family:

```
CREATE TABLE TWkeyspace.TWtable (id int PRIMARY KEY, column1 TEXT,
column2 TEXT,..., column19 TEXT)
```

After creating the key-space and column family, the Cassandra cluster is ready for the training workload.

5.2.3 Training Workloads

We defined training workloads in our experiment as mix of random read and random write operations. The write workload starts with an empty column family each time. Randomly generated string values are inserted into the fields using the statement:

```
INSERT INTO TWkeyspace.TWtable(id,column2,.. ,column19)
VALUES(key,value1,..,value19)
```

For the read operations, we allow the specification of the data that should be written before the actual workload. We generate data in the same manner used for the write workload. For the read requests, we use select point queries that retrieve one row by matching the key value. The following statement was used:

```
SELECT * FROM TWkeyspace.TWtable WHERE id=value
```

The search values, which are used in this query are generated using the same random generator, which was used for generating the data in the write operation. The latency of the Cassandra cluster is measured for a whole workload. That is, the time needed to answer all the requests of the workload is calculated and reported to a file. We varied the percentage of the write/read operations in the workloads. Our experiment included the following mixes: W10/R90, W30/R70, W40/R60, W50/R50, W70/R30, W90/R10, W100/R0, W0/R100.

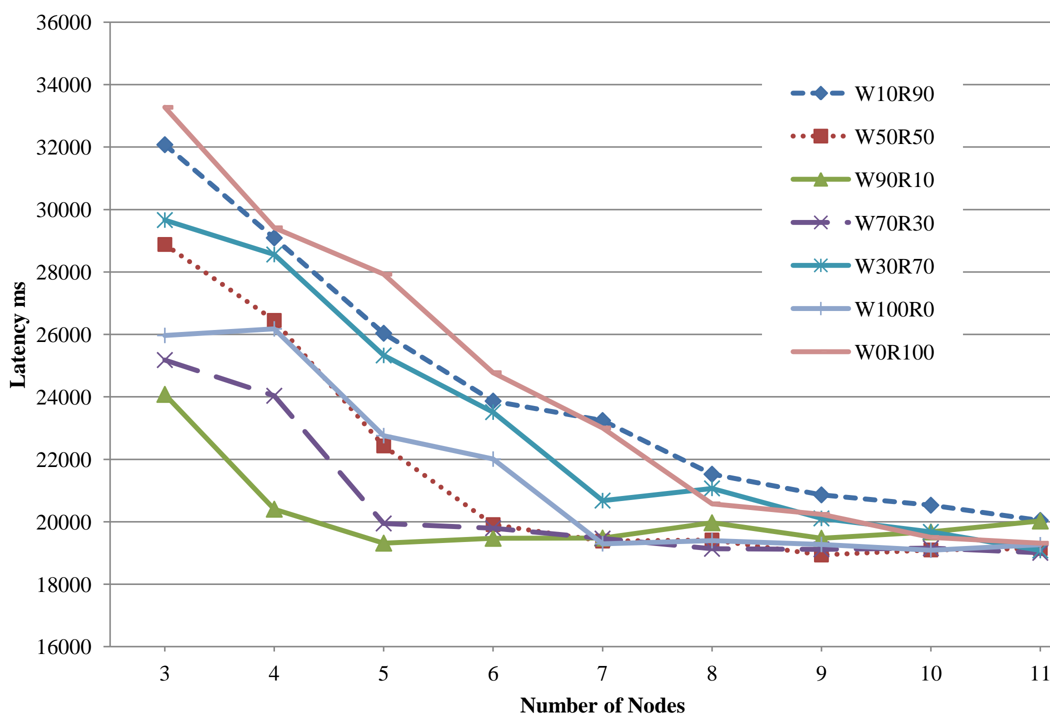
The training component ran on dedicated machines in the same network as the machines used for the data storage system installation. One machine ran the training component manager, whereas 3 machines were used to run the training component instances. We set the training component manager to start 5 Java clients on each of the 3 virtual machines. Each of these Java clients started 10 threads. The performance of the storage system is then measured by latency for the whole workload. The number of requests was 525k per workload. 90% of the generated data was used to create the prediction model, whereas the remaining 10% was used to validate prediction accuracy. Data generated from the training phase is presented and analyzed in the coming section.

5.3 Experimental Results

In this section, we provide the result from the training phase. Then we list the used modeling techniques and the result from the modeling phase. At the end, we illustrate how the framework can be used in a use case example.

5.3.1 Training Data and Model Generation

90% of the data that resulted from the training phase is the input for the analytical regression step. The data is of the form represented in the table 5.2b, which includes values of the measured latency. Figure 5.3 visualizes the change of latency of a Cassandra



(a) Latency of the Cassandra Cluster in Relation to Nodes Number

Nodes	3	4	5	6	7	8	9	10	11
W10R90	32075	29090	26037	23866	423242	21523	20862	20531	20043
W50R50	28881	26450	22429	19909	19382	19417	18936	19099	19144
W90R10	24080	20403	19316	19471	19479	19969	19474	19675	20024
W70R30	25179	24035	19946	19794	19462	19134	19118	19154	19007
W30R70	29662	28559	25328	23511	20678	21070	20108	19675	19080
W100R0	25968	26178	22759	22008	19295	19396	19273	19091	19265
W0R100	33271	29415	27927	24777	23004	20576	20237	19501	19314

(b) Training Data for the Regression Process

Figure 5.2: Training Data: Latency of the Cassandra Cluster with Different Workloads

cluster with different cluster sizes and different workloads identified by the write/read percentage. Cassandra was optimized by design for write operations [LM10], which explains why latency decreases for workloads with higher percentage of write operations compared to those with higher percentage of read operations. Of course, other factors affect the performance, such as the size of the cluster and the operations' throughput.

Two things should be taken into consideration when analyzing training data in order to generate a prediction model. The first one is that: "there is an infinite number of models, which will perfectly fit a finite data set" [KSV08, Kee11]. The second one is that: "All models are wrong, some are useful" [Box78, Box86]. What this means for us is that there are definitely several models that would fit the training data set. The question is then how to choose one that is, from one side, useful to guide the tuning process and easily created, on the other side.

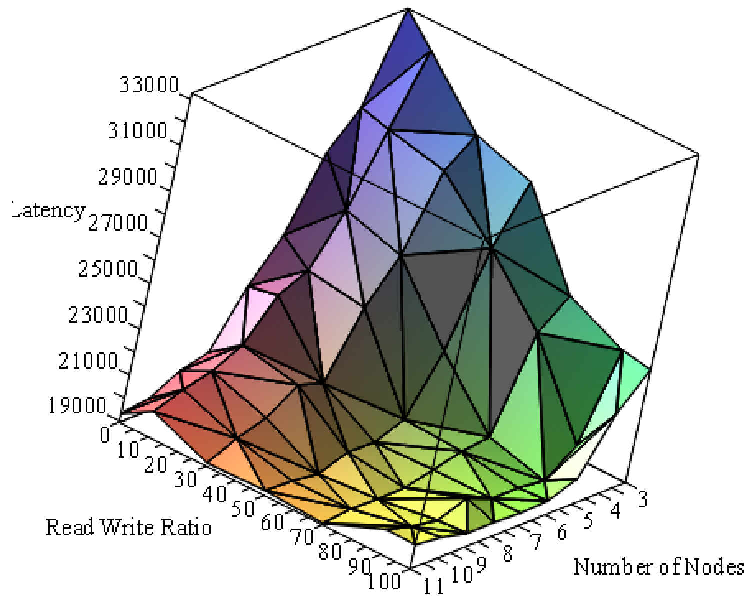


Figure 5.3: Latency of the Cassandra Cluster in Relation to the Number of Nodes & Write/Read Ratio

Consequently, we used a number of techniques to generate models that fit the training data. The choice among the models then was made based on finding the one that produces minimal prediction error [APB15]. The following regression techniques were used in the regression process:

- Simple Regression.
- Polynomial Regression with several degrees.
- Exponential Regression.

As a result from the regression process using the cubic regression gives the best residual standard deviation (i.e. difference between predicted and observed values) among the tested techniques, with a slight difference from the quadratic regression. In Figure 5.4, we illustrate the surface representing the resulted cubic model versus the data points representing the input measurements.

To validate the result from the regression process, we test the models' prediction power against the 10% of the training data, which was not used in its generation, and calculate the mean absolute error percentage. The cubic model gives high prediction accuracy of 96.4%. In Figure 5.5, we illustrate how the different considered models act. In this figure, we visualize for a workload with a write/read percentage W40/R60, as an example, the result from prediction and measurements. The curves represent the behavior as predicted using different models. The data points represent the real measurement of the Cassandra cluster latency when applying the workload. Both polynomial and

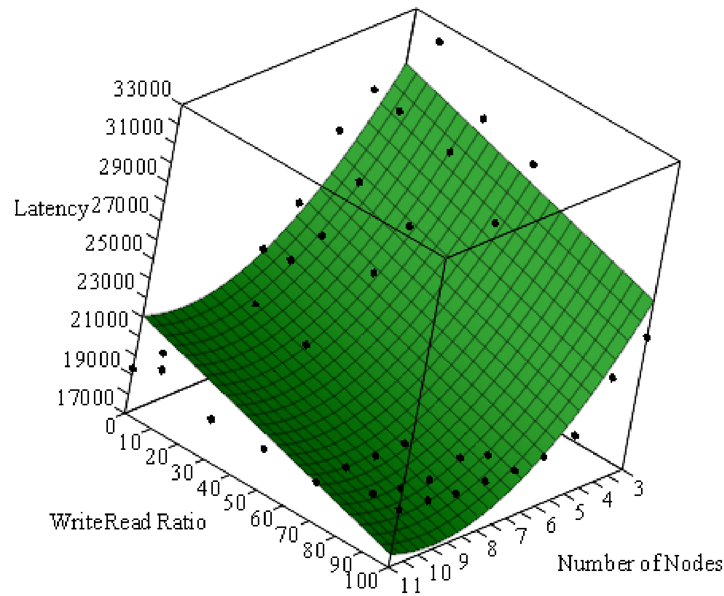
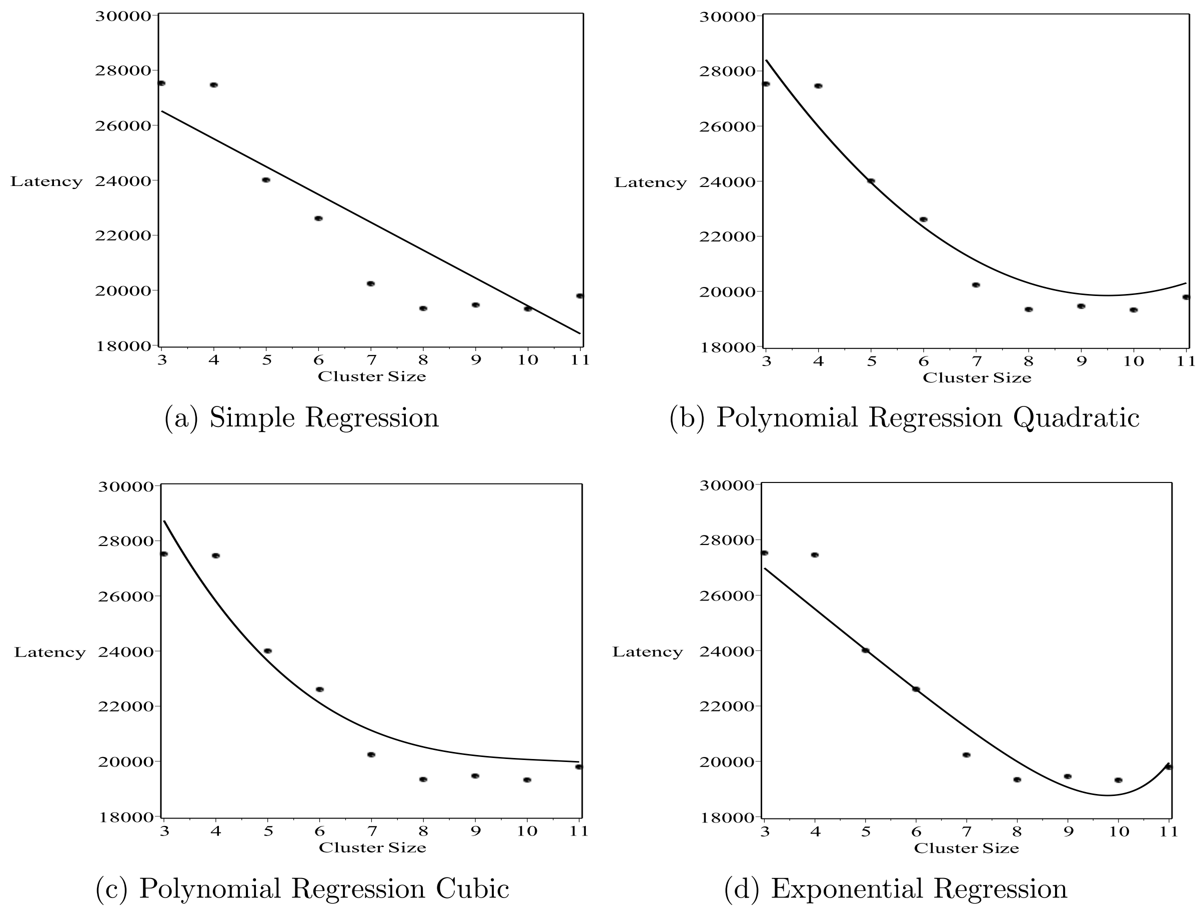


Figure 5.4: Input Measurements vs. Regression Analysis Result



(a) Simple Regression

(b) Polynomial Regression Quadratic

(c) Polynomial Regression Cubic

(d) Exponential Regression

Figure 5.5: Measurements vs. Prediction of Different Regression Techniques

exponential regression techniques model the performance within the tested range of cluster size.

The result from our experiment and evaluation shows that simple regression techniques are successful to provide a model, which characterizes the performance with high prediction accuracy. Such a model (even with the low number of independent parameters) can be beneficial to avoid allocating resources from which insignificant benefit is gained on the database cluster performance. Our training component allows specifying several workloads parameters, which allows extending the models. This would mean expanding the training phase to iterate (vary) the values of any new parameter that would be included.

5.3.2 The Search Algorithm Alternatives

For our evaluation, we want to search for the optimal alignment of sub-clusters that achieves minimal latency for workloads. In other words, the search algorithm is used to find the optimal cluster size that should be allocated to each workload, achieving a minimal latency for all.

In this section, we discuss the brute-force-based Algorithm 2, and the genetic Algorithm 3 validity as a search approach for the decision component. To evaluate both algorithms, we generate data representing the latency value for up to 5 workloads with cluster size that varies from 3 to 100 nodes. This data is used as input for both algorithms, which are evaluated with respect to accuracy and overhead.

Brute-force

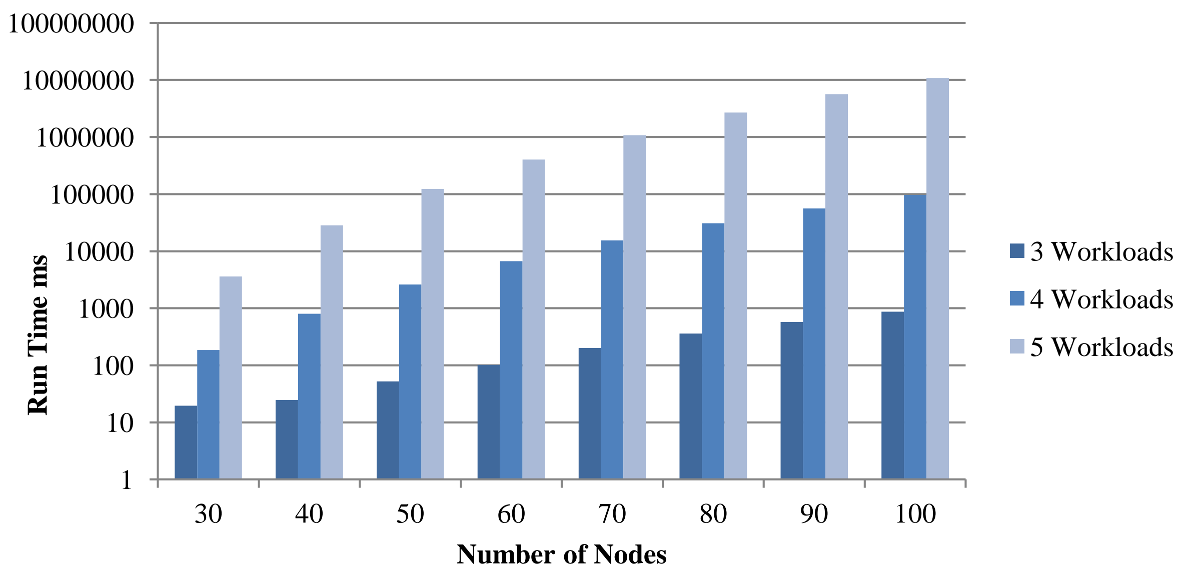


Figure 5.6: Runtime for the Decision Component using Brute-force

We use the brute-force algorithm to iterate over generated values of latency for considered workload/workloads with the different possibilities of cluster size. The brute-force algorithm is implemented in Java and run on a Windows 7 PC with 4GB memory, and 3GHz Intel CPU. Each run, was repeated 20 times and the average run time was considered. In Figure 5.6, we show the runtime of the decision component with different numbers of nodes and different number of workloads. As the measured runtime suggests, the search problem is polynomial to the number of nodes and exponential to the number of workloads. In the case of 5 workloads and 80 nodes, it takes the brute-force, to come up with the solution, 45 minutes. For a larger number of workloads, the run time exceeds an hour.

Genetic Algorithm

For the genetic algorithm, we start with an even distribution of nodes on workloads. While keeping the minimum number of nodes assigned to a workload not less than its replication factor, and the sum of nodes assigned to all workloads not more than all nodes, we randomly change the number of nodes. The execution time of the genetic algorithm is controlled by specifying the number of generations. For that reason, we focus on evaluating the quality of result of our approach while varying the number of generations. In Figure 5.7, we illustrate the result from running the genetic algorithm on 5 workloads with 100 nodes. We varied the number of generations from 10 to 100 and observed the value that resulted from the genetic algorithm compared to its initial value, and the optimal value. With 100 generations, we get a near optimal solution with an average runtime that does not exceed tens of ms.

The brute-force approach is straight forward and easy to implement. It always finds the optimal model-based solution. However, the disadvantage with brute-force is its cost,

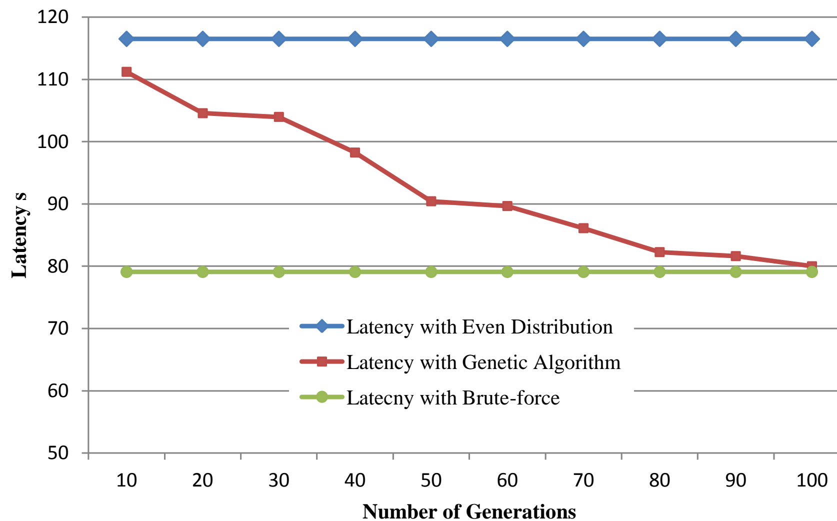
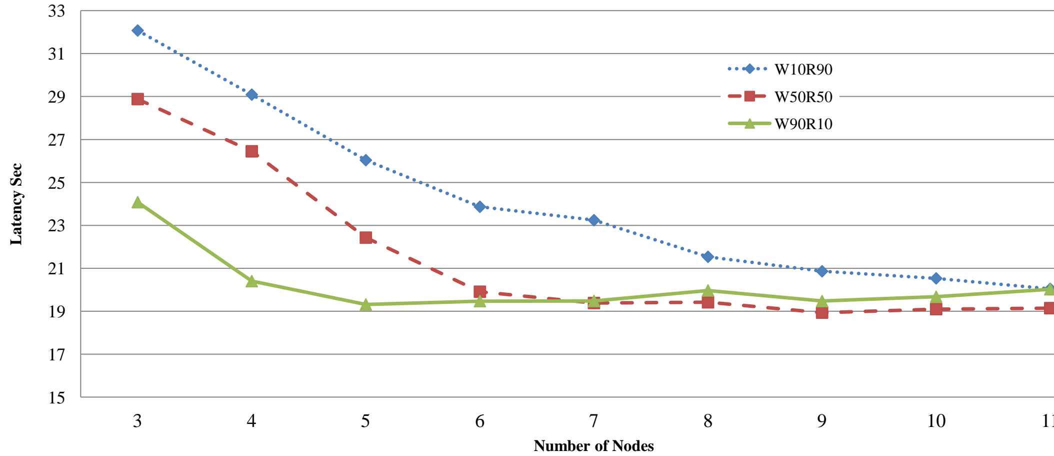


Figure 5.7: Genetic Algorithm Result



(a) Measured Latency Characteristics for Different Workloads

Number of Nodes	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Nodes for W10R90	3	3	4	3	4	5	5	6	6	8	8	9	9	10	11
Nodes for W50R50	3	3	3	5	5	5	6	6	6	6	6	6	7	7	7
Nodes for W90R10	3	4	4	4	4	4	4	4	5	4	5	5	5	5	5
Sum of Latency sec	85	81.4	78.3	74.9	71.9	68.9	66.3	64.2	63.1	61.8	60.8	60.1	59.6	59.2	58.7

(b) Optimal Allocation of Nodes to Workloads

Figure 5.8: Optimal Allocation of Nodes for Three Workloads

which is proportional to the search space. In our scenario, the search space is likely to grow especially when the models are extended with other workload, cluster, and storage system characteristics. Brute-force approach can be enhanced with dynamic programming. Dynamic programming approach saves processing time at the cost of more memory/storage space consumption [LM07].

The genetic algorithm approach that we used, finds a near optimal model-based solution in a small runtime compared to the brute-force. However, the algorithm that we used involves only the cluster size problem and needs extension to include other characteristics of an optimal cluster configuration.

5.3.3 Use Case Example

As a use case, we consider three workloads: read-heavy: 10% write and 90% read (W10R90), equally-mixed: 50% write and 50% read (W50R50), and write-heavy 90% write and 10% read (W90R10). Each workload involves 1.2 million request and works on a column family of 20 columns of type string.

The optimization decisions that can be made using our framework are:

- Finding the optimal cluster size for a single workload, indicated by minimum of latency.

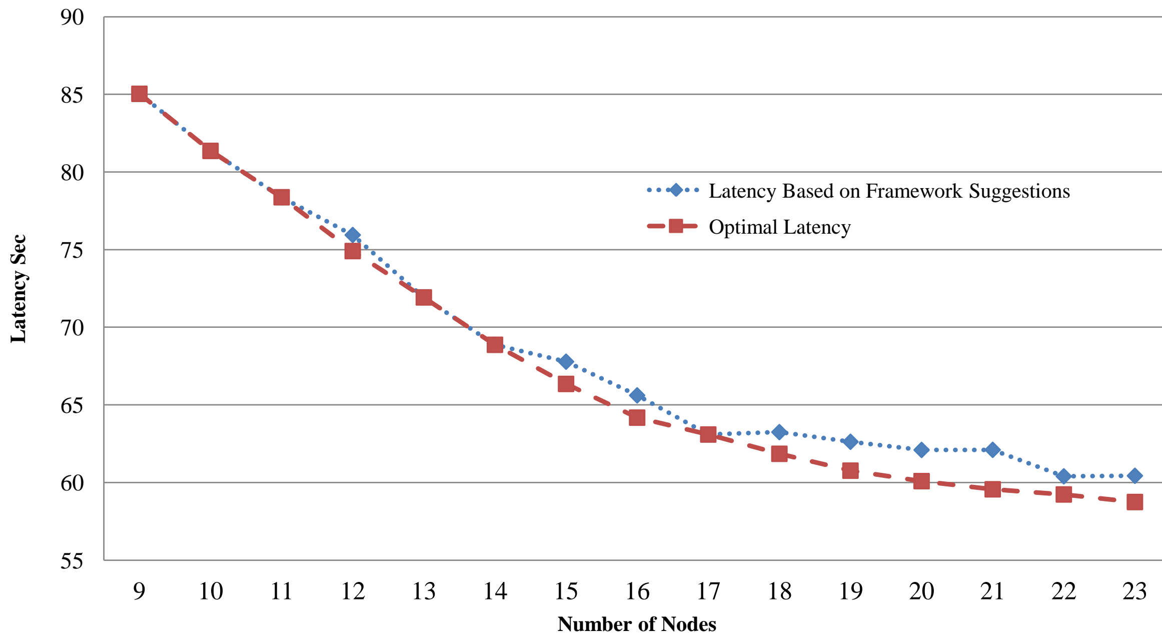


Figure 5.9: Framework Suggestion vs. Optimal Latency

- Creating an optimal setup of sub-clusters for all workloads, indicated by a global minimum of latency, and within resources restrictions.

We consider the second optimization task of using the framework to guide the process of finding the optimal allocation of nodes for the three workloads while achieving a minimal latency for all.

For a maximum of 23 nodes that we can allocate for the three workloads, the framework takes an average of less than 20 millisecond to find the solution of: 11 nodes for the first workload, 7 nodes for the second workload, and 5 for the third workload. Table 5.8b on the preceding page illustrates examples of the of optimal sub-cluster allocation for different maximum number of nodes. The sum of the overall latency was used as an optimization goal, though different aggregation functions are conceivable. For illustration purposes, the latency of the the three workloads is visualized in 5.8a on the previous page.

To evaluate the quality of the framework decision of optimal allocation of nodes, we run the workloads and measured the latency with the different cluster sizes. The sum of latency of workloads based on the framework suggestions compared to the optimal possible sum of latency is illustrated in Figure 5.9. As the figure show, the framework provides a near optimal decision of nodes allocation for the three workloads.

5.4 Summary

In the previous sections, we provided an overview of the used infrastructure and the Cassandra deployment for our experiments. A cluster of 11 virtual machines was ded-

icated for Cassandra deployment. Another 4 machines were used for the framework deployment and workloads generation.

To generate the training data, training workloads with varied percentages of write/read operations were generated while monitoring the latency of Cassandra for a whole workload. In each training phase the size of the dedicated cluster for the Cassandra deployment was varied between a minimum of 3 (the replication factor) and a maximum of 11 (the available nodes). 90% of the collected data was analyzed using different regression analysis techniques. The prediction power of the models was then measured using the remaining 10% of the data. The result from this phase shows that simple regression techniques are successful to provide a model which characterizes the performance with high accuracy.

Additionally, we evaluated the brute-force and the genetic-based algorithms as alternatives for the decision component. For that we generated data representing the latency value for up to 5 workloads with cluster size that varies from 3 to 100 nodes. This data was used as input for both algorithms, which we evaluated regarding result value and overhead. Though the brute force algorithm always finds the optimal solution, its overhead grows proportionally with the search space. The genetic algorithm based approach allows minimizing the overhead. However, the accuracy of the result depends on the number of generations.

Last, we provided a use case example for our framework. In this example, we applied Cassandra and used the framework for the problem of assigning an optimal number of nodes to various workloads running on sub-clusters to achieve the minimum overall latency.

6. Conclusion and Future Work

With the design and implementation of our framework, we improved the state of the art of (self-)tuning for Cloud data storage clusters. We demonstrated the benefit of the proposed framework in an experimental evaluation. Nevertheless there are still many aspects that can be improved for the overall framework and within the presented approaches.

In this chapter, we provide a summary of the previous chapters and outline the contribution of the thesis. Then, we introduce possibilities of future work.

6.1 Summary of the Dissertation

Cloud storage systems are increasing in popularity and complexity. Though these systems are designed to support self-management properties, such as elasticity, and availability, there is still a number of decisions to be made to actually fit the requirements of given applications to provide suitable performance. Because of the typical shared nothing architectures with data partitioning and replication, some performance aspects are addressed for the overall system. Nevertheless, the typical multi-layered distributed architecture of several component systems adds complexity to the tuning tasks. Moreover, if there are several applications with different and possibly changing workloads, using the same data storage cluster, there is little chance to tune for a specific application. Within the aforementioned scenario, we defined the tuning-problem as the following:

To find a Cloud storage system cluster configuration c out of a set of possible configurations CC that minimizes (assuming a standard form of the problem) the aggregated costs for all workloads w of a set of workloads WL that need to be supported by the overall cluster while still fulfilling their performance thresholds.

$$opt = \underset{c \in CC}{\text{minimize}} \quad \underset{w \in WL}{\Gamma} cost(w, c_w)$$

Gamma Γ represents an aggregation function suitable to the given cost components and the considered optimization goals, such as average, and sum.

To solve this problem, we introduced a self-tuning framework. This framework depends on predicting the performance of the underlying storage system in order to choose the cluster configuration that would achieve the optimization goals of all workloads.

We defined the input of the framework to be workload characteristics, infrastructure information, and optimization goals. The output is then defined as pairs of workload and storage cluster configuration. In order to predict the performance, we design the framework to build predictive models of performance in relation to infrastructure and workload characteristics. To achieve that, we chose the empirical data-driven approach for building predictive models.

From the architectural point of view, our framework consists of the following components:

- Training component: responsible for generating statistical data about the storage systems performance with different workloads and different cluster configurations.
- Cost estimation component: responsible for analyzing the training data using machine learning techniques in order to build a predictive model.
- Decision component: responsible for finding the cluster configuration that would achieve the optimization goals of the workloads based on model predicted values.
- Monitoring/refinement component: responsible for monitoring the performance and triggers a remolding process when the prediction model fails to provide the required accuracy.
- Knowledge-base: responsible for storing information for re-use by the framework.

The framework prototype is implemented and evaluated in the offline mode. Offline tuning mode allows informed configuration decisions and leads to better resource provisioning. The monitoring/refinement component, in this mode, is disabled. The tuning process, in this case, starts by a training phase that generates statistical data for creating the cost/performance models.

In our implementation, the training component is designed to run different training phases based on user-defined XML settings file. The current implementation supports the automatic generation of workloads containing mixes of random/sequential read, and write operations. After the training phase, gathered data is aggregated and analyzed to create the cost estimation models.

For the data analysis and modeling step, our implementation supports several regression analytics techniques. Generated models are evaluated based on their prediction power and the model with the lowest error rate is chosen.

Later on, the decision component finds the optimal nodes allocation for a set of input workloads while minimizing the sum of latency values for all workloads. In order to search for the optimal workload configuration, we discussed two approaches: brute-force based approach, and genetic-algorithm based approach.

For the evaluation of our framework, we performed several training phases where the cluster configuration and workload criteria were varied. We varied the number of nodes in Cloud storage cluster between the data replication factor and the maximum number of nodes in the available infrastructure. The workload and data that we used was syntactically generated by our framework and include varied mixes of random write and read operations.

We summarize the contribution of the thesis in the following:

- As a precondition for the proposed framework, we relate tasks of (self-)tuning to layers and sub-clusters within typical Cloud storage architecture. We define a guideline for typical Cloud storage system (self-)tuning processes.
- We build a training component that automates the process of testing and monitoring a Cloud storage cluster with different configurations and different workloads. This component generates training data needed to model the performance of Cloud storage cluster and can be used as a benchmark.
- We build a cost estimation component to predict performance metrics based on the statistical data driven approach. Our results show that typical regression analytic techniques provide good results.
- Based on measured and/or modeled performance of applications, we designed a decision component and evaluated two search algorithms; brute-force and genetic algorithm. Though the brute-force provides the optimal solution, its cost grows proportionally with the search space. Our genetic algorithm based approach provided near optimal solution with a smaller cost compared to the brute force approach.
- We address the essential problem of tuning the size of (sub-)clusters of the targeted workloads. Our framework supports creating an optimal setup of sub-clusters for all workloads, indicated by a global minimum of latency, and within resources restrictions.

6.2 Future Work

As more application exhibit unpredictable workloads, especially for Cloud-based data services, the (self-)tuning of storage systems is dramatically gaining importance [BCW09]. Moreover, the database systems and workloads have become so complex that tuning tools and self-management capabilities are not only desirable but necessary [CW09b].

During our work on a self-tuning framework for Cloud storage clusters, we came across many opportunities to extend the research in further directions. However, because of limited resources and time, we focused on the described area of the wide spectrum of possibilities. In this section, we provide an outline of possible future research directions in the following points:

Online refinement: our current implementation supports the framework offline mode. An important future direction is to implement the monitoring/refinement component to enable the model refinement and allow the framework to be reactive to workload changes.

Further performance metrics: in our implementation, we focused on minimizing the average latency of the system for a whole workload. We plan to extend our framework to support further performance metrics. A discussion of the requirements and possibilities of integrating other metrics (electricity consumption, and monetary costs) was presented earlier in the thesis.

Support multi-objective tuning process: this allows finding the optimal cluster configuration in the existence of tradeoff between contradicting goals, such as minimizing the cost e.g., energy consumption, while maximizing performance e.g., throughput. This requires extending the decision component with multi-objective optimization techniques.

Extend the model with further tuning knobs: our framework, by design, supports extending the model with several tuning knobs. This would lead to further research on different points. First, the modeling process should be revisited to verify if the used regression analysis techniques are sufficient. Second, the generated models should be analyzed to predict bottlenecks and identify tuning knobs, whose increase/decrease would optimally enhance the performance and minimize the costs.

Investigate the correlation between the tuning knobs: in our implementation, we treat tuning knobs as independent variables. However, the decisions made for one tuning knob can affect the possibilities and decisions that are made on other tuning knob. Thus, we find systematic research study on the correlations between the tuning knobs is required.

Extend the decision component and the search algorithm: as a result from extending the prediction model with further tuning knobs, the search space for the decision component grows. Thus, any brute-force approach would cause unacceptable overhead. The genetic algorithm approach presented in our work, provides a starting point for further extension.

Workload characterization: workload characteristics are assumed to be input for the framework in the design phase before deployment. However, workload characteristics are likely to change over time. We plan to investigate the area of

automatic identification and prediction of workload changes and adapting the underlying storage cluster according to the new needs.

Support for heterogeneous cluster nodes: our framework was designed with the assumption that nodes in the underlying cluster/infrastructure are homogenous with no difference in capabilities. This assumption is reasonable when using the framework in the design phase of a deployment cluster. However, as systems evolve, nodes with different computing/storage power are typically added. An interesting extension of our work is to expand the performance models to take nodes' heterogeneity into consideration.

A. Appendix

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:vc="http://www.w3.org/2007/XMLSchema-versioning"
   elementFormDefault="qualified" attributeFormDefault="unqualified"
   vc:minVersion="1.1">
      <xs:element name="TCSettings">
      <xs:complexType>
      <xs:sequence>
      <xs:element name="ClusterSetting">
      <xs:complexType>
      <xs:sequence>
      <xs:element name="Node" maxOccurs="unbounded">
      <xs:complexType>
      <xs:attribute name="IPAdress"/>
      <xs:attribute name="dataFolderPath"/>
      </xs:complexType>
      </xs:element>
      </xs:sequence>
      <xs:attribute name="cName"/>
      <xs:attribute name="totalNodesNum" type="xs:integer"/>
      </xs:complexType>
      </xs:element>
      <xs:element name="Database">
      <xs:complexType>
      <xs:sequence>
      <xs:element name="table" maxOccurs="unbounded">
      <xs:complexType>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="column">
      <xs:complexType>
      <xs:attribute name="name"/>
      <xs:attribute name="dataType"/>
      </xs:complexType>
      </xs:element>
      </xs:sequence>
      <xs:attribute name="tName"/>
      <xs:attribute name="columnNum" type="xs:integer"/>
      <xs:attribute name="rowNum" type="xs:integer"/>
      <xs:attribute name="sizeMB" type="xs:integer"/>
      </xs:complexType>
      </xs:element>
      </xs:sequence>
      <xs:attribute name="DPStrategy" type="xs:string"/>
      <xs:attribute name="keySpace" type="xs:string"/>
```

```
        <xs:attribute name="replicationFactor" type="xs:integer"/>
        </xs:complexType>
        </xs:element>
        <xs:element name="TrainingWorkload">
        <xs:complexType>
        <xs:attribute name="repeat" type="xs:integer"/>
        <xs:attribute name="measurementFolderPath" type="xs:string"/>
        <xs:attribute name="randomReadPercentage" type="xs:integer"/>
        <xs:attribute name="seqReadPercentage" type="xs:integer"/>
        <xs:attribute name="randomWritePercentage" type="xs:integer"/>
        <xs:attribute name="seqWritePercentage" type="xs:integer"/>
        <xs:attribute name="requestNum"/>
        </xs:complexType>
        </xs:element>
        <xs:element name="TISettings">
        <xs:complexType>
        <xs:sequence>
        <xs:element name="node" maxOccurs="unbounded">
        <xs:complexType>
        <xs:attribute name="IPAdress" type="xs:string"/>
        <xs:attribute name="clientNum"/>
        <xs:attribute name="threadsPerClient"/>
        </xs:complexType>
        </xs:element>
        </xs:sequence>
        </xs:complexType>
        </xs:element>
        </xs:sequence>
        </xs:complexType>
        </xs:element>
</xs:schema>
```

Listing A.1: Schema Definition of the Training Component Settings

```
<?xml version="1.0" encoding="UTF-8"?>
<TCSettings xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="bmSettingsSchema.xsd">
    <ClusterSetting cName="ClusterOne" totalNodesNum="11">
        <Node IPAdress="192.168.144.11"
          dataFolderPath="/home/training/cassandra"/>
        <Node IPAdress="192.168.144.12"
          dataFolderPath="/home/training/cassandra"/>
        <Node IPAdress="192.168.144.13"
          dataFolderPath="/home/training/cassandra"/>
        <Node IPAdress="192.168.144.14"
          dataFolderPath="/home/training/cassandra"/>
        <Node IPAdress="192.168.144.15"
          dataFolderPath="/home/training/cassandra"/>
        <Node IPAdress="192.168.144.16"
          dataFolderPath="/home/training/cassandra"/>
        <Node IPAdress="192.168.144.17"
          dataFolderPath="/home/training/cassandra"/>
        <Node IPAdress="192.168.144.18"
          dataFolderPath="/home/training/cassandra"/>
        <Node IPAdress="192.168.144.21"
          dataFolderPath="/home/training/cassandra"/>
        <Node IPAdress="192.168.144.22"
          dataFolderPath="/home/training/cassandra"/>
        <Node IPAdress="192.168.144.23"
          dataFolderPath="/home/training/cassandra"/>
    </ClusterSetting>
    <Database DPStrategy="SimpleStrategy" keySpace="sibaKS"
      replicationFactor="3">
        <table tName="users">
            <column name="id" dataType="uuid"/>
            <column name="user_name" dataType="ascii"/>
            <column name="password" dataType="ascii"/>
            <column name="email" dataType="ascii"/>
            <column name="gender" dataType="text"/>
            <column name="phoneNum" dataType="text"/>
            <column name="country" dataType="text"/>
            <column name="birthY" dataType="bigint"/>
            </table>
    </Database>
    <TrainingWorkload repeat="20" seqWritePercentage="100"
      requestNum="520000"
      measurementFolderPath="/home/training/measurement"/>
    <TISettings>
```

```
<node IPAdress="192.168.144.24" clientNum="3"
    threadsPerClient="5"/>
<node IPAdress="192.168.144.25" clientNum="3"
    threadsPerClient="5"/>
<node IPAdress="192.168.144.26" clientNum="3"
    threadsPerClient="5"/>
</TISettings>
</TCSettings>
```

Listing A.2: Training Component Settings

Bibliography

[Aba09] Daniel Abadi. Data Management in the Cloud: Limitations and Opportunities. *Data Engineering Bulletin*, 32(1):3–12, 2009. (cited on Page 25)

[ABPA+09] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In *In Proceedings of Conference on Very Large Databases VLDB*, pages 922–933. VLDB Endowment, August 2009. (cited on Page 8, 13, and 65)

[ADEA12] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. *Data Management in the Cloud Challenges and Opportunities*. Morgan and Claypool Publishers, 2012. (cited on Page 1, 7, and 9)

[APB15] Anestis Antoniadis, Jean-Michel Poggi, and Xavier Brossat. *Modeling and Stochastic Learning for Forecasting in High Dimensions*. Springer, 2015. (cited on Page 53, 75, and 91)

[Apl13] P. Aplin. Benchmarking Cassandra on Violin. Technical Report 1.0, Locomatix inc, 2013. (cited on Page 32 and 34)

[ASS+09] Ashraf Aboulnaga, Kenneth Salem, Ahmed A. Soror, Umar Farooq Minhas, Peter Kokosielis, and Sunil Kamath. Deploying Database Appliances in the Cloud. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 32(1):13–20, 2009. (cited on Page 26)

[BCD+11] Philip A. Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya, David B. Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. Adapting Microsoft SQL Server for Cloud Computing. In *Proceedings of the 27th International Conference on Data Engineering (ICDE)*, number 1255–1263, pages 1255–1263. IEEE, April 2011. (cited on Page 1)

[BCW09] Nicolas Bruno, Surajit Chaudhuri, and Gerhard Weikum. *Encyclopedia of Database Systems*, chapter Database Tuning using Online Algorithms, pages 741–744. Springer US, 2009. (cited on Page 101)

[BKKL09] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. How is the Weather Tomorrow?: Towards a Benchmark for the Cloud. In *Proceedings of the Second International Workshop on Testing Database Systems*, pages 9:1–9:6. ACM, June 2009. (cited on Page 30)

[BLL+14] Amir Hossein Borhani, Philipp Leitner, Bu-Sung Lee, Xiaorong Li, and Terence Hung. WPress: An Application-Driven Performance Benchmark for Cloud-Based Virtual Machines. In *the 18th International Enterprise Distributed Object Computing Conference (EDOC)*, pages 101–109. IEEE, Sept 2014. (cited on Page 32 and 34)

[BMB12] Tom Bostoen, Sape Mullender, and Yolande Berbers. Analysis of Disk Power Management for Data-Center Storage Systems. In *Procedings of the 3rd International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet (e-Energy)*, pages 1–10. IEEE, May 2012. (cited on Page 24)

[Box78] George E. P. Box. Robustness in the Strategy of Scientific Model Building. In *the Proceedings of Robustness in Statistics Workshop*, pages 201–236, April 1978. (cited on Page 90)

[Box86] George E. P. Box. *Empirical Model-Building and Response Surfaces*. John Wiey & Sons, 1986. (cited on Page 90)

[Bre10] Spell T. Brett. *Pro Java Programming*. Apress, 2010. (cited on Page 64)

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *Transactions on Computer Systems*, 26(2):4:1–4:26, 2008. (cited on Page 1, 7, 10, 11, 17, 18, 31, 33, 34, 64, 73, and 87)

[CLS10] Anderson J. Chris, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide*. O'Reilly Media, Inc., 2010. (cited on Page 15)

[CMH11] Yun Chi, Hyun Jin Moon, and Hakan Hacigümüş. iCBS: Incremental Cost-based Scheduling Under Piecewise Linear SLAs. In *Proceedings of the VLDB Endowment*, pages 563–574. VLDB Endowment, June 2011. (cited on Page 26)

[CMT00] Maria Calzarossa, Luisa Massari, and Daniele Tessera. Workload Characterization Issues and Methodologies. In *Performance Evaluation: Origins and Directions*, pages 459–481. Springer, June 2000. (cited on Page 42)

[CRB+11a] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. CloudSim: a Toolkit for Modeling and Simulation

of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011. (cited on Page 29)

[CRB11b] Rodrigo N. Calheiros, Rajiv Ranjan, and Rajkumar Buyya. Virtual Machine Provisioning Based on Analytical Performance and QoS in Cloud Computing Environments. In *the International Conference on Parallel Processing (ICPP)*, pages 295–304. IEEE, September 2011. (cited on Page 29)

[CRS+08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008. (cited on Page 1, 7, 14, and 31)

[CS93] Maria Calzarossa and Giuseppe Serazzi. Workload Characterization: a Survey. *Proceedings of the IEEE*, 81(1136–1150):8, 1993. (cited on Page 42)

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154. ACM, June 2010. (cited on Page 30, 31, 32, 33, 34, 45, 68, 73, and 87)

[CW06] Surajit Chaudhuri and Gerhard Weikum. Foundations of Automated Database Tuning. Online, April 2006. Published in ICDE 2006; Available online at http://icde06.ewi.utwente.nl/cwicde06.pdf; visited on June 24th,2015. (cited on Page 21 and 22)

[CW09a] Surajit Chaudhuri and Gerhard Weikum. *Encyclopedia of Database Systems*, chapter Database Tuning using Trade-off Elimination, pages 744–748. Springer US, 2009. (cited on Page 22)

[CW09b] Surajit Chaudhuri and Gerhard Weikum. *Encyclopedia of Database Systems*, chapter Self-Management Technology in Databases, pages 2550–2555. Springer US, 2009. (cited on Page 101)

[Cyr02] Michele Cyran. *Oracle9i Database Concepts*. Oracle Corporation, 2002. (cited on Page 13 and 14)

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008. (cited on Page 11, 17, and 65)

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. *Symposium on Operating Systems Principles*, 41(6):205–220, 2007. (cited on Page 1, 7, 10, 14, and 15)

[DMRT11] Thibault Dory, Boris Mejías, Peter Van Roy, and Nam Luc Tran. Comparative Elasticity and Scalability Measurements of Cloud Databases. In *Proceedings of the Second Symposium on Cloud Computing SOCC*. ACM, October 2011. (cited on Page 32 and 34)

[DMT11] Kamal Dahbur, Bassil Mohammad, and Ahmad Bisher Tarakji. A Survey of Risks, Threats and Vulnerabilities in Cloud Computing. In *Proceedings of the International Conference on Intelligent Semantic Web-Services and Applications*, pages 12:1–12:6. ACM, April 2011. (cited on Page 24)

[DPCCM13] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013. (cited on Page 31)

[FE10] Borko Furht and Armando Escalante. *Handbook of Cloud Computing*. Springer, 2010. (cited on Page 16 and 17)

[FGC+97] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 78–91. ACM, Oct 1997. (cited on Page 17)

[FPR12] Michael Frank, Meikel Poess, and Tilmann Rabl. Efficient Update Data Generation for DBMS Benchmarks. In *Proceedings of the 3rd International Conference on Performance Engineering*, pages 169–180. ACM, April 2012. (cited on Page 31)

[FZRL08] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *Grid Computing Environments Workshop, GCE 08*, pages 1–10. IEEE, November 2008. (cited on Page 5)

[GGK+14] Andrea Gandini, Marco Gribaudo, WilliamJ. Knottenbelt, Rasha Osman, and Pietro Piazzolla. Performance Evaluation of NoSQL Databases. In *Computer Performance Engineering*. Springer, 2014. (cited on Page 30)

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 29–43. ACM, Oct 2003. (cited on Page 1, 7, 10, and 11)

[GLN+12] Íñigo Goiri, Kien Le, Thu D. Nguyen, Jordi Guitart, Jordi Torres, and Ricardo Bianchini. GreenHadoop: Leveraging Green Energy in Data-processing Frameworks. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 57–70. ACM, April 2012. (cited on Page 24 and 25)

[GR13] John Gantz and David Reinse. The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and the Biggest Growth in the Far East. In IDC, February 2013. (cited on Page 16)

[GRH+13] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. BigBench: Towards an Industry Standard Benchmark for Big Data Analytics. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1197–1208. ACM, June 2013. (cited on Page 31 and 34)

[Här87] Theo Härder. Realisierung von operationalen Schnittstellen. In *Datenbank Handbuch*, pages 163–335. Springer, 1987. (cited on Page 9 and 20)

[HDBD09] Parag H. Dave and Himanshu B. Dave. *Design and Analysis of Algorithms*. Pearson Education India, 2009. (cited on Page 81)

[Hel07] Pat Helland. Life beyond Distributed Transactions: an Apostate's Opinion. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 132–142. ACM, Jan 2007. (cited on Page 7)

[Hew10] Eben Hewitt. *Cassandra: The Definitive Guide*. O' Reilly Media, 2010. (cited on Page 14)

[Hil05] Jane Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 2005. (cited on Page 26)

[HLL+11] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma B. Cetin, and Shivnath Babu. Starfish: A Self-tuning System for Big Data Analytics. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 261–272, January 2011. (cited on Page 26)

[HMS11] David Hand, Heikki Mannila, and Padhraic Smyth. *Principles of Data Mining*. MIT Press, 2011. (cited on Page 29 and 52)

[JCR11] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic Optimization for MapReduce Programs. *Proceedings of the VLDB Endowment*, 4(6):385–396, 2011. (cited on Page 66)

[Kee11] Karel J. Keesman. *System Identification: An Introduction*. Springer, 2011. (cited on Page 29 and 90)

[KKR14] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. Benchmarking Scalability and Elasticity of Distributed Database Systems. *Proceedings of the VLDB Endowment*, 7(12):1219–1230, 2014. (cited on Page 33, 34, and 45)

[KN10] Gunnar Kreitz and Fredrik Niemela. Spotify – Large Scale, Low Latency, P2P Music-on-Demand Streaming. In *International Conference on Peer-to-Peer Computing (P2P)*, pages 1–10. IEEE, Aug 2010. (cited on Page 64)

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992. (cited on Page 53)

[KSV08] Mark Kotanchek, Guido Smits, and Ekaterina Vladislavleva. Trustable Symbolic Regression Models: Using Ensembles, Interval Arithmetic and Pareto Fronts to Develop Robust and Trust-aware Models. In *Genetic Programming Theory and Practice V*, Genetic and Evolutionary Computation Series, pages 201–220. Springer, 2008. (cited on Page 76 and 90)

[KYTA12] 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. IEEE, April 2012. (cited on Page 42)

[LLH+11] Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang. YSmart: Yet Another SQL-to-MapReduce Translator. In *Proceedings of the 31st International Conference on Distributed Computing Systems*, pages 25–36. IEEE, Jun 2011. (cited on Page 11)

[LM07] Art Lew and Holger Mauch. *Dynamic Programming: A Computational Tool*. Springer, 2007. (cited on Page 95)

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *Operating Systems Review*, 44(2):35–40, 2010. (cited on Page 10, 14, 18, 64, and 90)

[MBS12] Siba Mohammad, Sebastian Breß, and Eike Schallehn. Cloud Data Management: A Short Overview and Comparison of Current Approaches. In *In Proceedings of the 24th Workshop Grundlagen von Datenbanken*, pages 41–46. CEUR-WS, May 2012. (cited on Page 5 and 9)

[MF10] Andréa Matsunaga and José A. B. Fortes. On the Use of Machine Learning to Predict the Time and Resources Consumed by Applications. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 495–504. IEEE, May 2010. (cited on Page 29)

[MG11] Peter Mell and Timothy Granc. The NIST Definition of Cloud Computing Recommendation of the National Institute of Standard and Technology. online, September 2011. (cited on Page 1 and 5)

[MHCD10] Asit K. Mishra, Joseph L. Hellerstein, Walfredo Cirne, and Chita R. Das. Towards Characterizing Cloud Backend Workloads: Insights from Google Compute Clusters. *SIGMETRICS Performance Evaluation Review*, 37(4):34–41, 2010. (cited on Page 42)

[Mit96] Melanie Mitchell. *Computer Algorithms : Introduction to Design and Analysis*. MIT Press, 1996. (cited on Page 81)

[MSB13] Siba Mohammad, Eike Schallehn, and Sebastian Breß. Clustering the Cloud: A Model for (Self-)Tuning of Cloud Data Management Systems. In *Proceedings of the 3rd International Conference on Cloud Computing and Services Science (CLOSER)*, pages 520–524. SciTePress Science and Technology Publications, May 2013. (cited on Page 37 and 63)

[MSS15] Siba Mohammad, Eike Schallehn, and Gunter Saake. A Self-Tuning Framework for Cloud Storage Clusters. In *19th East-European Conference on Advances in Databases and Information Systems (ADBIS)*, pages 351–364. Springer, Sep 2015. (cited on Page 37, 63, and 85)

[Nee15] Nishant Neeraj. *Mastering Apache Cassandra*. Packt Publishing Ltd, 2015. (cited on Page xiv, 15, and 65)

[Nel91] Neal Nelson. A Benchmark Based on the Realities of Business. In *The Benchmark Handbook*. Morgan Kaufmann, 1991. (cited on Page 73)

[OK12] Rasha Osman and William J. Knottenbelt. Database System Performance Evaluation Models: A Survey. *Performance Evaluation*, 69(10):0166–5316, 2012. (cited on Page xiii, 27, and 29)

[PCC+11] Jordà Polo, Claris Castillo, David Carrera, Yolanda Becerra, Ian Whalley, Malgorzata Steinder, Jordi Torres, and Eduard Ayguadé. Resource-aware Adaptive Scheduling for Mapreduce Clusters. In *Proceedings of the 12th International Conference on Middleware*, pages 187–207. Springer, December 2011. (cited on Page 26 and 66)

[PPR+09] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A Comparison of Approaches to Large-scale Data Analysis. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 165–178. ACM, July 2009. (cited on Page 33)

[PS15] Alexander Pokluda and Wei Sun. Benchmarking Failover Characteristics of Large Scale Data Storage Applications: Cassandra

and Voldemort. Website, November 2015. Available online at http://www.alexanderpokluda.ca/coursework/cs848/CS848%20Project%20Report%20-%20Alexander%20Pokluda%20and%20Wei%20Sun.pdf. (cited on Page 34)

[Raa93] Francois Raab. TPC-C - The Standard Benchmark for Online transaction Processing (OLTP). In *The Benchmark handbook for Database and transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993. (cited on Page 73)

[RGVS+12] Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, Victor Muntés-Mulero, Hans-Arno Jacobsen, and Serge Mankovskii. Solving Big Data Challenges for Enterprise Application Performance Management. *Proceedings of the VLDB Endowment*, 5(12):1724–1735, 2012. (cited on Page 31, 32, 33, and 34)

[SB02] Dennis Shasha and Philippe Bonnet. *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*. Morgan Kaufmann, 2002. (cited on Page 19)

[Sch12] Eike Schallehn. Database Tuning and Self-Tuing. Online, 2012. Lecture Notes, available online at http://wwwiti.cs.uni-magdeburg.de/iti_db/lehre/advdb/SoSe2012/tuning.pdf; visited on June 24th, 2015. (cited on Page xiii, 19, and 22)

[Ser91] Omri Serlin. The History of DebitCredit and the TPC. In *The Benchmark Handbook*. Morgan Kaufmann, 1991. (cited on Page 73)

[SFKS10] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 2010. (cited on Page 13 and 14)

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies*, pages 1–10. IEEE, May 2010. (cited on Page 1, 7, 10, and 15)

[SLMBA11] Sherif Sakr, Anna Liu, Daniel M. Batista, and Mohammad Alomari. A Survey of Large Scale Data Management Approaches in Cloud Environments. *Communications Surveys & Tutorials*, 13(3):311–336, 2011. (cited on Page 6 and 7)

[SMA+08] Ahmed A. Soror, Umar Farooq Minhas, Ashraf Aboulnaga, Kenneth Salem, Peter Kokosielis, and Sunil Kamath. Automatic Virtual Machine Configuration for Database Workloads. *Transactions on Database Systems (TODS)*, 35(1):7:1–7:47, 2008. (cited on Page 29)

[SMK+11] Muhammad Bilal Sheikh, Umar Farooq Minhas, Omar Zia Khan, Ashraf Aboulnaga, Pascal Poupart, and David J. Taylor. A Bayesian Approach to Online Performance Modeling for Database Appliances Using Gaussian Models. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, pages 121–130. ACM, June 2011. (cited on Page 52)

[SMZ+10] Yingjie Shi, Xiaofeng Meng, Jing Zhao, Xiangmei Hu, Bingbing Liu, and Haiping Wang. Benchmarking Cloud-based Data Management Systems. In *Proceedings of the Second International Workshop on Cloud Data Management*, pages 47–54. ACM, Oct 2010. (cited on Page 25, 33, and 34)

[SSA08] Dimitri P. Solomatine, Linda M. See, and Robert J. Abrahart. Data-Driven Modelling: Concepts, Approaches and Experiences. *Practical Hydroinformatics*, 68:17–30, 2008. (cited on Page xiii, 28, and 51)

[SSH08] Gunter Saake, Kai-Uwe Sattler, and Andreas Heuer. *Datenbanken Konzepte und Sprachen.* Mitp-Verlag, 2008. (cited on Page 9 and 20)

[The05] The IBM Development Team. An Architectural Blue Print for Autonomic Computing. Online, June 2005. Available online at http://www-03.ibm.com/autonomic/pdfs/ACBlueprintWhitePaperV7.pdf; Visited on June 24th, 2015. . (cited on Page xiii, 22, 23, and 43)

[The15a] The Amazon Development Team. Amazon S3. Website, Apr 2015. Available online at http://aws.amazon.com/s3/; Visited on April 23rd, 2015. (cited on Page 10)

[The15b] The Amazon Development Team. Amazon SimpleDB. Website, Apr 2015. Available online at http://aws.amazon.com/simpledb/; Visited on April 23rd, 2015. (cited on Page 10)

[The15c] The Amazon S3 Development Team. Amazon S3 Design Requirements and Principles. Website, June 2015. Available online at http://www.amazon.com/gp/node/index.html?ie=UTF8&merchant=&node=16427261; Visited in June 4th, 2015. (cited on Page 7)

[The15d] The Cassandra Development Team. Cassandra Query Language. Website, Apr 2015. Available online at https://cassandra.apache.org/doc/cql3/CQL.html; Visited on April 23rd, 2015. (cited on Page 11)

[The15e] The Datastax Development Team. Cassandra Stress Tool. Website, May 2015. Available online at http://docs.datastax.com/en/cassandra/1.2/cassandra/tools/toolsCStress_t.html; Visited on May 13th, 2015. (cited on Page 29)

[The15f] The HBase Development Team. Apache HBase. Website, Apr 2015. Available online at http://hbase.apache.org/; Visited on April 23rd, 2015. (cited on Page 18 and 64)

[Tho00] Alexander Thomasian. Performance Analysis of Database Systems. In *Performance Evaluation: Origins and Directions*, pages 305–327. Springer, 2000. (cited on Page 27)

[TSJ+09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A Warehousing Solution over a Map-reduce Framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009. (cited on Page 11, 13, 14, and 64)

[VKJ14] Carlos Vazquez, Ram Krishnan, and Eugene John. Cloud Computing Benchmarking: A Survey. In *Proceedings of the International Conference on Grid Computing and Applications (GCA)*, pages 1–6. Springer, July 2014. (cited on Page 45)

[WLZZ14] Huajin Wang, Jianhui Li, Haiming Zhang, and Yuanchun Zhou. Benchmarking Replication and Consistency Strategies in Cloud Serving Databases: HBase and Cassandra. In *Big Data Benchmarks, Performance Optimization, and Emerging Hardware*. Springer, 2014. (cited on Page 33, 34, and 45)

[WWRL10] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-Preserving Public Auditing for Data Storage Security in Cloud Computing. In *In the Proceedings of INFOCOM*, pages 1–9. IEEE, March 2010. (cited on Page 25)

[XCZ+11] Pengcheng Xiong, Yun Chi, Shenghuo Zhu, Hyun Jin Moon, Calton Pu, and Hakan Hacigümüş. Intelligent Management of Virtualized Resources for Database Systems in Cloud Environment. In *Proceedings of the 27th International Conference on Data Engineering*, pages 87–98. IEEE, April 2011. (cited on Page 26)

[XP09] Kaiqi Xiong and Harry Perros. Service Performance and Analysis in Cloud Computing. In *the World Conference on Services - I*, pages 693 – 700. IEEE, July 2009. (cited on Page 29)

[YKGS12] Rerngvit Yanggratoke, Gunnar Kreitz, Mikael Goldmann, and Rolf Stadler. Predicting Response Times for the Spotify Backend. In *Proceedings of the 8th International Conference on Network and Service Management*, pages 117–125. IEEE, October 2012. (cited on Page 27 and 30)

[ZCB10] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud Computing: State of the Art and Research Challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010. (cited on Page xiii, 6, and 24)

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, pages 15–28. USENIX Association, April 2012. (cited on Page 65)

[ZKJ+08] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 29–42. USENIX Association, December 2008. (cited on Page 66)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den April 22, 2016