SLA-Driven Database Replication on Cloud Platforms

by

Liang Zhao

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

FACULTY OF ENGINEERING

THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY · AUSTRALIA

Tuesday 14th May, 2013

PLEASE TYPE

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: Zhao	
First name: Liang	Other name/s:
Abbreviation for degree as given in the University calendar: PhD	
School: School of Computer Science and Engineering	Faculty: Faculty of Engineering
Title: SLA-Driven Database Replication on Cloud Platforms	

Abstract 350 words maximum: (PLEASE TYPE)

Rapidly growing Internet-based services have substantially redefined the way of providing data persistence and retrieval from that of the one-size-fits-all solution offered by relational database management systems to a full spectrum of cloud databases solutions. This significant paradigm shift did not happen spontaneously, but its progress and adoption was hastened by the boom in cloud computing adoption. Cloud computing also represents a new resource provisioning paradigm that shifts the location of resources to the network to reduce the costs associated with the management of hardware and software resources.

This thesis takes the unique cloud platform customer's perspective and explores in detail the trade-off characteristics between performance gain and monetary cost across different cloud platforms. These related problems are studied: 1) generic performance evaluations of different cloud providers; 2) the service level agreement (SLA) gaps between the cloud providers and the cloud customers.

The design and architecture of cloud varies among cloud providers. For performance evaluations, this thesis spends the initial two chapters on addressing two generic evaluation solutions for different cloud platforms and cloud databases respectively. The first solution proposes a generic evaluation framework that focuses on performance, availability, and reliability characteristics of various cloud platforms. The second solution provides a generic benchmark architecture for benchmarking cloud databases, specifically NoSQL database as a service. It measures the performance of replication delay and monetary cost.

As existing SLAs of cloud providers guarantee only the availability of their services, rather than supporting the straightforward requirements and restrictions under which SLAs of cloud customers' applications need to be handled, this thesis uses another two chapters to further investigate the approach for the customer's interest to automate SLA-driven management for database replication on virtualized database servers. The investigation takes two steps. In the first step, the performance of database replication of virtualized database servers are comprehensively evaluated. The second step takes the lesson learned from the first step to build a SLA-driven framework for managing database replication. The framework implements customer-centric dynamic provisioning mechanisms for virtualized database servers based on adaptive application requirements.

赵亮	[illegible]	Tuesday 23rd April, 2013
Signature	Witness	Date

ORIGINALITY STATEMENT

Signed …………[illegible]………………..............

Date ……………Tuesday 23rd April, 2013……………..............

To Mom, Dad, and Jessie...

Abstract

Rapidly growing Internet-based services have substantially redefined the way of providing data persistence and retrieval from that of the one-size-fits-all solution offered by relational database management systems to a full spectrum of cloud databases solutions. This significant paradigm shift did not happen spontaneously, but its progress and adoption was hastened by the boom in cloud computing adoption. Cloud computing also represents a new resource provisioning paradigm that shifts the location of resources to the network to reduce the costs associated with the management of hardware and software resources.

This thesis takes the unique cloud platform customer's perspective and explores in detail the trade-off characteristics between performance gain and monetary cost across different cloud platforms. These related problems are studied: 1) generic performance evaluations of different cloud providers; 2) the service level agreement (SLA) gaps between the cloud providers and the cloud customers.

The design and architecture of cloud varies among cloud providers. For performance evaluations, this thesis spends the initial two chapters on addressing two generic evaluation solutions for different cloud platforms and cloud databases respectively. The first solution proposes a generic evaluation framework that focuses on performance, availability, and reliability characteristics of various cloud platforms. The second solution provides a generic benchmark architecture for benchmarking cloud databases, specifically NoSQL database as a service. It measures the performance of replication delay and monetary cost.

As existing SLAs of cloud providers guarantee only the availability of their services, rather than supporting the straightforward requirements and restrictions under which SLAs of cloud customers' applications need to be handled, this thesis uses another two chapters to further investigate the approach for the customer's interest to automate SLA-driven management for database replication on virtualized database servers. The investigation takes two steps. In the first step, the performance of database replication of virtualized database servers are comprehensively evaluated. The second step takes the lesson learned from the first step to build a SLA-driven framework for managing database replication. The framework implements customer-centric dynamic provisioning mechanisms for virtualized database servers based on adaptive application requirements.

Acknowledgments

First and foremost, I am sincerely grateful to Dr. Anna Liu for supervising me. I thank her for guiding and inspiring me constantly through the entire period of this research. I have especially benefited from her insight, positive attitude, breadth of knowledge, and continuous encouragement. Without her grateful support throughout the school and NICTA since my first research project back when I was still pursuing my master's degree, I would not have been able to achieve this success.

I would like to acknowledge the guidance of my co-supervisors, Dr. Sherif Sakr and Dr. Hye-Young Helen Paik. I thank Dr. Sherif Sakr for his invaluable guidance, constructive feedback, and patient support throughout my research in NICTA. I would also like to thank him for motivating me and always being available to talk about research. I thank Dr. Hye-Young Helen Paik for her generous support in the school. I would also like to thank her for introducing me to my first research project when I was pursuing the master degree.

I would also like to thank Dr. Jacky Keung for his research collaboration in the work presented in Chapter 3. I would like to thank Prof. Alan Fekete, Dr. Hiroshi Wada, and Kevin Lee for their contributions in Chapter 4 and Chapter 5. I would like to thank Dr. Hiroshi Wada, Dr. Liming Zhu and Dr. Xiwei Xu for their works in related to Chapter 6. I would also like to thank Dr. Rajiv Ranjan for his collaboration in the work (Ranjan and Zhao, 2011; Ranjan et al., 2010). I sincerely thank Dr. Wykeen Seet for spending time on proofreading this thesis and providing all comments.

My special thanks to the researchers in the group: Dr. Hiroshi Wada and Kevin Lee, for their encouragement and help through the difficult times. More special thanks go to the group members: Dr. Xiwei Xu, Dr. Yin Kia Chiam, Thi Khanh Van Tran, Qinghua Lu, Paul Rimba, Sadeka Islam, Matthew Sladescu, Huangtao Xiong, and Fei Teng, for their remarkable companionship. I really enjoy working and sharing the fun time with them. I am grateful to the staff members in the school and NICTA for helping me in many different ways. The gratefulness goes to Smart Services CRC and NICTA for providing the scholarship, which enables me to live and conduct this research in Australia. The gratefulness also goes to Amazon for providing AWS in Education Research Grant which supports many experiments conducted in this thesis.

Last, but most importantly, from the bottom of my heart, I would like to thank my parents, Lijuan Chu and Dianchi Zhao, for their constant love, support and encouragement during my Ph.D. study. I would also like to express my appreciation to my girlfriend, Huan Wang, for her patience, understanding, and encouragement. The work in this thesis would not have been possible without their support and care.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Over the past decade, rapidly growing Internet-based services have substantially redefined the way of providing data persistence and retrieval from that of the one-size-fits-all solution offered by relational database management systems to a full spectrum of cloud databases solutions. This significant paradigm shift did not happen spontaneously, but its progress and adoption was hastened by the boom in cloud computing adoption. Cloud computing technology also represents a new paradigm for the provisioning of computing resources. This paradigm shifts the location of resources to the network to reduce the costs associated with the management of hardware and software resources. Therefore, it promises a number of advantages for the deployment of data-intensive applications, such as elasticity of resources, pay-per-use cost model, low time to market, and the perception of unlimited resources and infinite scalability. Hence, it is now possible, at least theoretically, to achieve unlimited throughput by continuously adding computing resources if the workload increases.

Cloud computing is by its nature a virtualized and shared environment where its design and architecture varies among cloud providers. From the cloud customer's perspective, these characteristics result in concerns on the differences of performance gain and monetary cost of different cloud platforms, and the different cloud databases which run on top the platform. Motivated by this, this thesis spends two chapters on addressing two generic performance evaluation solutions for different cloud platforms and cloud databases respectively.

The first solution proposes a generic evaluation framework which focuses on performance, availability, and reliability characteristics of various cloud platforms. The framework is implemented with a unified interface for different cloud platforms. It generates high stress or low stress load on the cloud platforms to measure throughput and response time in three scenarios, namely end-user – cloud host, cloud host – cloud database, and end-user – cloud database. Three cloud platforms, including Amazon Web Services, Microsoft Windows Azure, and Google App Engine have been examined by implementing the framework on each platform. Further analysis of errors and faults is carried out based on the results to explore the availability and reliability of the three cloud platforms.

The second solution provides a generic benchmark architecture for benchmarking cloud databases, specifically NoSQL database as a service. It measures the performance of replication delay and monetary cost. The implementations of the architecture have been used to benchmark several NoSQL database as a service offerings, including Amazon SimpleDB and S3, Microsoft Windows Azure Table Storage and Blob Storage, and Google App Engine Datastore. The performance changes of replication delay have been discovered on SimpleDB. And trade-off analysis has been performed with regards to response time, throughput and monetary cost.

The performance variation of different cloud platforms is not the only concern from the cloud customers' perspective, even within a single cloud platform supported by a cloud provider, Cooper et al. (2010); Schad et al. (2010) have also reported that the variation of the performance is high due to the nature of resource sharing. These observations raise concerns of the service level agreements (SLAs) that the cloud customers can offer to their end-users. As existing SLAs by cloud providers are not designed for supporting the straightforward requirements and restrictions under which SLAs of cloud customers' applications need to be handled, most providers guarantee only the availability of their services (Suleiman et al., 2012). Therefore, customer concerns on SLA handling for their cloud applications and their cloud databases, especially for those of virtualized database servers which are simply ported from a conventional data center into the cloud. Motivated by this, this thesis uses another two chapters to investigate the approach for the customer's interest to automate SLA-driven management for database replication on virtualized database servers.

The investigation takes two steps. In the first step, the performance of database replication of virtualized database servers are comprehensively evaluated, with MySQL running on Amazon EC2 as an example. The idea of performing the evaluation of database replication of NoSQL database as a service is adopted and transformed to suit that of virtualized database servers. It is implemented with a Web 2.0 application and tested with several choices of database distributions, including same zone, different zone and different region. With the observation of throughput and replication delay, the trade-off for master-slave replication strategy is addressed next.

The second step takes the lesson learned from the first step to build a SLA-driven framework for managing database replication. The framework implements customer-centric dynamic provisioning mechanisms for virtualized database servers based on adaptive application requirements. The framework, as a middleware, continuously monitors the database workload, tracks the satisfaction of the application-defined SLA, evaluates the condition of the action rules and takes the appropriate actions when necessary.

1.2 Contributions

The main contributions of this thesis can be summarized on chapter basis as follows:

A general framework for performance evaluation of cloud platforms

- Design and development of a novel architecture runtime evaluation framework for cloud platforms from cloud customers' perspective, called CARE. It is designed to use a unified interface for different cloud platforms, such as Amazon Web Services, Microsoft Windows Azure, and Google App Engine, therefore allowing direct performance comparison where, before, it was simply not possible to do.
- A comprehensive collection of results from conducting cloud performance experiments over several cloud platforms with a number of test scenarios and test loads from the framework, that show what are the performance, availability, and reliability characteristics of different cloud platforms.
- A study on the exceptions and error analysis based on empirical results to show the reasons behind the faults and errors.
- A summary of development challenges that customers, such as developers and architects, could face when using cloud platforms as their production environment for service delivery.

Performance evaluation of database replication of NoSQL database as a service

- Design and development of a simple, but effective architecture of benchmarking database replication of NoSQL database as a service. Several implementations of this architecture are deployed for Amazon SimpleDB and S3, Microsoft Windows Azure Table Storage and Blob Storage, and Google App Engine Datastore.

- A collection of detailed measurements over several storage platforms, that show how frequently, and in what circumstances, different inconsistency situations are observed, and its impact on customer observable performance properties from choosing to operate with weak consistency mechanisms.
- A study on the trade-offs in monetary cost or performance to compensate different consistency options in different NoSQL database as a service.

Performance evaluation of database replication of virtualized database servers

- Design and development of a customized Cloudstone benchmark for benchmarking database replication of virtualized database servers. The benchmark is implemented and deployed in Amazon EC2.
- An alternative measurement and calculation approach was proposed to reduce variation (standard deviation) of delay measurement by an order of magnitude. In particular, it alleviates the problem of inaccurate replication delay measurement and calculation that is caused by a fast clock drift phenomenon in Amazon EC2.
- A study on the limits to scaling for an application that itself manages virtualized database replica servers in the cloud with the benchmark was conducted. In particular, the average replication delay and throughput that could exist with an increasing number of virtualized database replica servers and different configurations to the geographical locations was measured.
- Identification of the trade-offs of load on the master copy, the workload imposed on each slave copy when processing updates from the master, and also from the increasing staleness of replicas.

A framework of SLA-driven database replication on virtualized database servers

- A presentation of an end-to-end framework for customer-centric SLA management of virtualized database servers is provided. The framework facilitates adaptive and dynamic provisioning of the database tier of the software applications based on application-defined policies for satisfying their own SLA performance requirements, avoiding the cost of any SLA violation and controlling the monetary cost of the allocated computing resources.
- A demonstration of the experimental results with geographic distributed virtualized database servers to show the effectiveness of the SLA-based framework in providing the customer applications with the required flexibility for achieving their SLA requirements is also provided.

1.3 Publications

This thesis is based on a series of refereed research papers. The logical mapping between the chapters and the number of involved papers can be summarized in Table 1.1.

And the detailed information of each involved paper are listed as follows:

1. Rajiv Ranjan, Liang Zhao, Xiaomin Wu, Anna Liu, Andres Quiroz and Manish

Table 1.1: The logical mapping between the chapters and the number of involved papers

Chapters	Involved papers
Chapter 2	[1], [4], [9], [10] and other backgrounds
Chapter 3	[2]
Chapter 4	[3]
Chapter 5	[6]
Chapter 6	[5], [7], [8], and [9]

Parashar. Peer-to-peer cloud provisioning: service discovery and load-balancing. In *Cloud Computing: Computer Communications and Networks*, pages 195–217, 2010. Springer London. Available at: `http://dx.doi.org/10.1007/978-1-84996-241-4_12`.

2. Liang Zhao, Anna Liu and Jacky Keung. Evaluating cloud platform architecture with the CARE framework. In *Proceedings of the 17th Asia Pacific Software Engineering Conference*, APSEC '10, pages 60–69, Sydney, NSW, Australia, 2010. IEEE Computer Society. Available at: `http://dx.doi.org/10.1109/APSEC.2010.17`.

3. Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee and Anna Liu. Data consistency properties and the trade-offs in commercial cloud storages: the consumers perspective. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, CIDR '11, pages 134–143, Asilomar, CA, USA, 2011. Available at: `http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper15.pdf`.

4. Liang Zhao, Sherif Sakr and Anna Liu. *On the spectrum of web scale data management*, *Cloud Computing: Methodology, Systems, and Applications*, pages 487–509. CRC, 2011. Available at: `http://dx.doi.org/10.1201/b11149-25`.

5. Sherif Sakr, Liang Zhao, Hiroshi Wada and Anna Liu. CloudDB AutoAdmin: towards a truly elastic cloud-based data store . In *Proceedings of the IEEE 9th International Conference on Web Services*, ICWS '11, pages 732–733, Washington, DC, USA, 2011. IEEE Computer Society. Available at: `http://dx.doi.org/10.1109/ICWS.2011.19`.

6. Liang Zhao, Sherif Sakr, Alan Fekete, Hiroshi Wada and Anna Liu. Application-managed database replication on virtualized cloud environments. In *Proceedings of the IEEE 28th International Conference on Data Engineering Workshops*, ICDEW '12, Washington, DC, USA, 2012. IEEE Computer Society. Available at: `http://dx.doi.org/10.1109/ICDEW.2012.77`.

7. Liang Zhao, Sherif Sakr and Anna Liu. Application-managed replication controller for cloud-hosted databases. In *Proceedings of the IEEE 5th International Conference on Cloud Computing*, IEEE CLOUD '12, pages 922–929, Honolulu, HI, USA, 2012. IEEE Computer Society. Available at: `http://dx.doi.org/10.1109/CLOUD.2012.35`.

8. Liang Zhao, Sherif Sakr, Liming Zhu, Xiwei Xu and Anna Liu. An architecture framework for application-managed scaling of cloud-hosted relational databases. In *Proceedings of the WICSA/ECSA 2012 Companion Volume*, WICSA/ECSA '12, pages 21–28, Helsinki, Finland, 2012. ACM. Available at: `http://dx.doi.org/10.1145/2361999.2362004`.

Moreover, during the Ph.D. study, there are two journal papers published out of above papers. They are:

9. Rajiv Ranjan and Liang Zhao. Peer-to-peer service provisioning in cloud computing environments. *J. Supercomput.*, Online First, Oct. 2011. Available at: `http://dx.doi.org/10.1007/s11227-011-0710-5`.

10. Liang Zhao, Sherif Sakr and Anna Liu. A framework for consumer-centric SLA management of cloud-hosted databases. *IEEE Trans. Serv. Comput.*, PrePrint, Feb. 2013. Available at: `http://dx.doi.org/10.1109/TSC.2013.5`.

1.4 Thesis organization

Chapter 2 provides a literature review of work in the area that is related to this thesis in three parts. The first part provides a complete overview of cloud computing and also discusses the state-of-the-art of a few public cloud platforms. The second part two provides an overview of cloud databases. It starts with concepts, challenges, and trade-offs of cloud databases in general, and ends with a broad survey of the state-of-the-art of public cloud databases in three categorizations. Part two also pays extra attentions on the NoSQL movement and the stat-of-the-art of NoSQL database systems. The third part describes the challenges of SLA management for virtualized database servers and the main research aim of this thesis.

Chapter 3 addresses the performance evaluation problem on cloud platforms. There have been a number of research efforts that specifically evaluated the Amazon cloud platform. However, there has been little in-depth evaluation research conducted on other cloud platforms, such as Google App Engine and Microsoft Windows Azure. But more importantly, these work lack a more generic evaluation method that enables a fair comparison between the various cloud platforms. Motivated by this, in this thesis a novel approach called CARE, Cloud Architecture Runtime Evaluation, is developed to perform four test set methods with different load stresses against cloud hosting servers or cloud databases from the perspective of the end-user or the cloud host. The framework is capable to address performance, availability, and reliability characteristics of various cloud platforms. The overall data analysis of faults and errors based on intensive collected data, for deducing architecture internal insights, is also another contribution.

Chapter 4 investigates the replication evaluation on NoSQL database as a service. NoSQL database as a service is part of the database as a service offering to complement traditional database systems by rejecting of general ACID transactions as one common feature. NoSQL database as a service has been supported by many service providers that offer various consistency options, from eventual consistency to single-entity ACID. With different consistency options, the correlated performance gains are unclear to many customers. Therefore, in this thesis a simple benchmark is proposed for evaluating replication delay of NoSQL database as a service from the customers' perspective. The detailed measurements over several NoSQL database as a services offerings show how frequently, and in what circumstances, different inconsistency situations are observed, and to what impact the customers sees on performance characteristics from choosing to operate with weak consistency mechanisms. The overall methodology of experiments, for measuring consistency from a customer's view, is also another contribution.

Chapter 5 describes a solution to replication evaluation on virtualized database servers. In addition to the two widespread approaches, namely NoSQL database as a service and relational database as a service, virtualized database servers is the third approach for deploying data-intensive applications in cloud platforms. It takes advantages of virtualization technologies by taking an existing application designed for a conventional data center and then porting it to virtual machines in the public cloud. Such migration process usually requires minimal changes in the architecture or the code of the deployed application. In this thesis, the limits to scaling for an application that itself manages database replicas in virtualized database servers in the cloud is explored. A few important limits are characterized in the load on the master copy, the workload imposed on each slave copy when processing updates from the master, and also from the increasing staleness of replicas.

Chapter 6 introduces a SLA-driven framework for managing database replication. Cloud-hosted database systems, such as virtualized database servers, powering cloud-hosted applications form a critical component in the software stack of these applications. However, the specifications of existing SLA for cloud services are not designed to flexibly handle even relatively straightforward performance and technical requirements of customer applications. Motivated by this, in this thesis a novel adaptive approach for SLA-based management of virtualized database servers from the customer perspective is presented. The framework is database platform-agnostic, supports virtualized database servers, and requires zero source code changes of the cloud-hosted software applications. It facilitates dynamic provisioning of the database tier in software stacks based on application-defined policies for satisfying their own SLA performance requirements, avoiding the cost of any SLA violation and controlling the monetary cost of the allocated computing resources. Therefore, the framework is able to keep several virtualized database replica servers in different data centers to support the different availability, scalability and performance improvement goals. The experimental results confirm the effectiveness of the SLA-based framework in providing the customer applications with the required flexibility for achieving their SLA requirements.

Chapter 7 gives the conclusions of this thesis as well as some future research directions.

Chapter 2

Background and related work

Over the past decade, rapidly growing Internet-based services have substantially redefined the needs and approaches of data persistence and retrieval. Relational database management systems are not the one-size-fits-all solution anymore due to new challenges of ever-increasing needs for scalability and new application requirements. More and more specialized solutions are proposed to overcome the dissatisfaction on the inability of traditional databases. Together with one-size-fits-all solution, they have composed a full spectrum of cloud databases solutions, introducing a significant paradigm shift in database management. Nevertheless, this paradigm did not happen spontaneously, but is associated with the boom of cloud computing adoption.

Cloud computing technology represents a new paradigm for the provisioning of computing resources. This paradigm shifts the location of resources to the network to reduce the costs associated with the management of hardware and software resources. It represents the long-held dream of envisioning computing as a utility (Armbrust et al., 2010) where the economy of scale principles help to effectively drive down the cost of computing resources. Cloud computing simplifies the time-consuming processes of hardware provisioning, hardware purchasing and software deployment. Therefore, it promises a number of advantages for the deployment of data-intensive applications, such as elasticity of resources, pay-per-use cost model, low time to market, and the perception of unlimited resources and infinite scalability. Hence, it becomes possible, at least theoretically, to achieve unlimited throughput by continuously adding computing resources if the workload increases.

It is impossible to take advantage of cloud databases without understanding cloud computing well. Therefore, this chapter first gives a complete overview of cloud computing from the perspectives of key definitions, related technologies, service and deployment models, and use cases in Section 2.1, followed by Section 2.2 which analyzes state-of-the-art of current public cloud computing platforms, with focus on their provisioning capabilities. Section 2.3 discusses an overview of cloud databases in regard to related concepts of cloud databases, as well as database management challenges and trade-offs in cloud. Section 2.4 describes state-of-the-art of NoSQL in detail, followed by a more broad survey of state-of-the-art of public cloud databases. In Section 2.6, the challenges of SLA management for virtualized database servers and the main research aim of this thesis will be discussed.

2.1 Overview of cloud computing

2.1.1 Definitions

Cloud computing is an emerging trend that leads to the next step of computing evolution, building on decades of research in virtualization, autonomic computing, grid computing, and utility computing, as well as more recent technologies in networking, web, and software services (Vouk, 2008). Although cloud computing is widely accepted nowadays, the definition of cloud computing is still arguable, due to the diversity of technologies composing the overall view of cloud computing.

Indeed, from the research perspective, many researchers have proposed their definitions of cloud computing by extending the scope of their own research domains. From the view of service-oriented architecture, Vouk (2008) implied cloud computing as "a service-oriented architecture, reduced information technology overhead for the end-user, greater flexibility, reduced total cost of ownership, on-demand services, and many other things". Buyya et al. (2009) derived the definition from clusters and grids, acclaiming for the importance of service-level agreements (SLAs) between the service provider and customers, describing that cloud computing is "a type of parallel and distributed system consisting of a collection of interconnected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resource(s) based on SLAs". Armbrust et al. (2010) from Berkeley highlighted three aspects of cloud computing including illusion of infinite computing resources available on demand, no up-front commitment, and pay-per-use utility model, arguing that cloud computing "consists of the service applications delivered over the Internet along with the data center hardware and systems software that provide those services". Moreover, from the industry perspective, more definitions and excerpts by industry experts can be categorized from the perspectives of scalability, elasticity, business models, and others (Vaquero et al., 2008).

It is hard to reach a singular agreement upon the definition of cloud computing, because of not only a fair amount of skepticism and confusion caused by various technologies, but also the prevalence of marketing hype. For that reason, National Institute of Standards and Technology has been working on proposing a guideline of cloud computing. The definition of cloud computing in the guideline has received fairly wide acceptance. It is described as "a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" (Mell and Grance, 2011). According to the definition, cloud computing can be identified with the following five essential characteristics, namely on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service.

2.1.2 Related technologies in cloud computing

Cloud computing has evolved out of decades of research in different related technologies from which it has inherited some features and functionalities such as virtualized environments, autonomic computing, grid computing, and utility computing. The Table 2.1 provides a summary of the feature differences between those technologies and cloud computing in short, while details of related technologies are discussed as following (Zhang et al., 2010):

Virtualization

Virtualization is a technology that isolates and abstracts the low-level resources and provides virtualized resources for high-level applications. In the context of hardware virtualization, the details of physical hardware can be abstracted away with support of hypervisors, such as Linux Kernel-based Virtual Machine[1], VMWare Eastic Sky X[2], and Xen[3]. A virtualized server managed by the hypervisor is commonly called a virtual machine. In general, several virtual machines can be abstracted from a single physical machine. With clusters of physical machines, hypervisors are capable of abstracting and pooling resources, as well as dynamically assigning or reassigning resources to virtual machines on-demand. Therefore, virtualization forms the foundation of cloud computing. Since a virtual machine is isolated from both the underlying hardware and other virtual machines. Providers can customize the platform to suit the needs of the customers by either

[1]`http://www.linux-kvm.org/`

[2]`http://www.vmware.com/products/vi/esx/`

[3]`http://xen.org/`

Table 2.1: Feature similarities and differences between related technologies and cloud computing

Related technologies	Differences	Similarities
Virtualization	Cloud computing is not only about virtualizing resources, but also about intelligently allocating resources for managing competing resource demands of the customers.	Both isolate and abstract the low-level resources for high-level applications.
Autonomic computing	The objective of cloud computing is focused on lowering the resource cost rather than to reduce system complexity as it is in autonomic computing.	Both interconnect and integrate distributed computing systems.
Grid computing	Cloud computing however also leverages virtualization to achieve on-demand resource sharing and dynamic resource provisioning.	Both employ distributed resources to achieve application-level objectives.
Utility computing	Cloud computing is a realization of utility computing.	Both offer better economic benefits.

exposing applications running within virtual machines as services, or providing direct access to virtual machines thereby allowing customers to build services with their own applications. Moreover, cloud computing is not only about virtualizing resources, but also about intelligent allocation of resources for managing competing resource demands of the customers.

Autonomic computing

Autonomic computing aims at building computing systems capable of self-management, which means being able to operate under defined general policies and rules without human intervention. The goal of autonomic computing is to overcome the rapidly growing complexity of computer system management, while being able to keep increasing interconnectivity and integration unabated (Kephart and Chess, 2003). Although cloud computing exhibits certain similarities to automatic computing the way that it interconnects and integrates distributed data centers across continents, its objective somehow is to lower the resource cost rather than to reduce system complexity.

Grid computing

Grid computing is a distributed computing paradigm that coordinates networked resources to achieve a common computational objective. The development of grid computing was originally driven by scientific applications which are usually computation-intensive, but applications requiring the transfer and manipulation of a massive quantity of data was also able to take advantage of the grids (Guo et al., 2010; Habib et al., 2006; Lehman et al., 2006). Cloud computing appears to be similar to grid computing in the way that it also employs distributed resources to achieve application-level objectives. However, cloud computing takes one step further by leveraging virtualization technologies to achieve on-demand resource sharing and dynamic resource provisioning.

Utility computing

Utility computing represents the business model of packaging resources as a metered services similar to those provided by traditional public utility companies. In particular, it allows provisioning resources on demand and charging customers based on usage rather than a flat rate. The main benefit of utility computing is better economics. Cloud computing can be perceived as a realization of utility computing. With on-demand resource provisioning and utility-based pricing, customers are able to receive more resources to handle unanticipated peaks and only pay for resources they needed; meanwhile, service providers can maximize resource utilization and minimize their operating costs.

2.1.3 Cloud service models

The categorization of three cloud service models defined in the guideline are also widely accepted nowadays. The three service models are namely *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS), and *Software as a Service* (SaaS).

As shown in Figure 2.1, the three service models form a stack structure of cloud computing, with Software as a Service on the top, Platform as a Service in the middle, and Infrastructure as a Service at the bottom, respectively. While the inverted triangle shows the possible proportion of providers of each model, it is worth mentioning that definitions of three service models from the guideline paid more attentions to the customers' view. In contrast, Vaquero et al. (2008) defined the three service models from the perspective of the providers' view. The following definitions of the three models combines the two perspectives (Mell and Grance, 2011; Vaquero et al., 2008), in the hope of showing the whole picture.

1. *Infrastructure as a Service*: Through virtualization, the provider is capable of splitting, assigning, and dynamically resizing the cloud resources including processing, storage, networks, and other fundamental computing resources to build virtualized systems as requested by customers. Therefore, the customer is able to deploy and run arbitrary operating systems and applications. The customer does not need to deploy the underlying cloud infrastructure but has control over which operating systems, storage options, and deployed applications to deploy with possibly limited control of select networking components. The typical providers are Amazon Elastic Compute Cloud (EC2)[4] and GoGrid[5].

2. *Platform as a Service*: The provider offers an additional abstraction level, which is a software platform on which the system runs. The change of the cloud resources

[4]`http://aws.amazon.com/ec2/`

[5]`http://www.gogrid.com/`

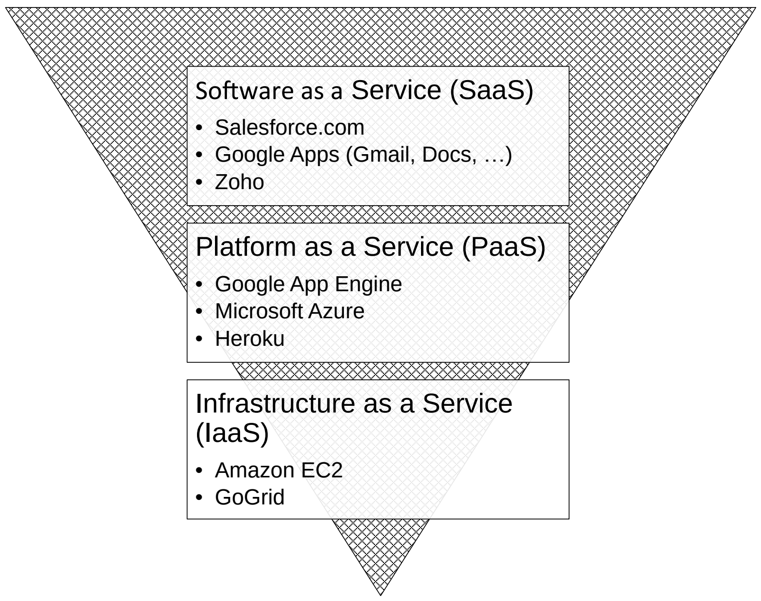

Software as a Service (SaaS)
• Salesforce.com
• Google Apps (Gmail, Docs, …)
• Zoho
Platform as a Service (PaaS)
• Google App Engine
• Microsoft Azure
• Heroku
Infrastructure as a Service (IaaS)
• Amazon EC2
• GoGrid

Figure 2.1: The service models of cloud computing

including network, servers, operating systems, or storage is made in a transparent manner. The customer does not need to deploy the cloud resources, but has control over the deployed applications and possibly application hosting environment configurations. Three platforms are well-known in this domain, namely Google App Engine[6], Microsoft Windows Azure Platform[7], and Heroku[8] which is a platform built on top of Amazon EC2. The first one offers Python, Java, and Go as programming platforms. The second one supports languages in .NET Framework, Java, PHP, Python, and Node.js. While the third one is compatible with Ruby, Node.js, Clojure, Java, Python, and Scala.

3. *Software as a Service*: The provider provides services of potential interest to a wide variety of customers hosted in its cloud infrastructure. The services are accessible from various client devices through a thin client interface such as a web browser. The customer does not need to manage the cloud resources or even indi-

[6]http://developers.google.com/appengine/
[7]http://www.windowsazure.com/
[8]http://www.heroku.com/

vidual application capabilities. The customer could, possibly, be granted limited user-specific application configuration settings. A variety of services, operating as Software as a Service, are available in the Internet, including Salesforce.com[9] (Weissman and Bobrowski, 2009), Google Apps[10], and Zoho[11].

2.1.4 Cloud deployment models

The guideline also defines four types of cloud deployment models (Mell and Grance, 2011), which are described as follows:

1. *Public cloud* offers infrastructures to be accessed by the general public via Internet. It may be managed by a third party service provider. And it exists on the premises of the service provider.

2. *Private cloud* offers similar advantages of public cloud, with better management, security, and resiliency characteristics. It is usually served within the organization to secure data and processes safely.

3. *Community cloud* shares infrastructures across a group of organizations. It is restricted to be manageable and controllable by the group members.

4. *Hybrid cloud* is a combination of two or more distinct cloud infrastructures, to leverage their respective benefits of data security and resiliency.

Table 2.2 summarizes the four cloud deployment models in terms of ownership, customership, location, and security.

[9] `http://salesforce.com/`
[10] `http://www.google.com/apps/`
[11] `http://www.zoho.com/`

Table 2.2: Summary of cloud deployment models

Deployment model	Ownership	Customership	Infrastructure location to customers	Security	Examples
Public cloud	Organization(s)	General public customers	Off-premises	No fine-grained control	Amazon Web Services
Private cloud	An organization/ A third party	Customers within an organization	On/Off-premises	Highest degree of control	Internal cloud platform to support business units in a large organization
Community cloud	Organization(s) in a community/ A third party	Customers from organizations that have shared concerns	On/Off-premises	Shared control among organizations in a community	Healthcare cloud for exchanging health information among organizations
Hybrid cloud	Composition of two or more from above	Composition of two or more from above	On/Off-premises	Tighter control, but require careful split between distinct models	Cloud bursting for load balancing between cloud platforms

2.2 Public cloud platforms: state-of-the-art

Key players in public cloud computing domain including Amazon Web Services, Microsoft Windows Azure, Google App Engine, Eucalyptus[12], and GoGrid offer a variety of prepackaged services for monitoring, managing, and provisioning resources. However, the techniques implemented in each of these clouds do vary.

For Amazon EC2, the three Amazon services, namely Amazon Elastic Load Balancer[13], Amazon Auto Scaling[14], and Amazon CloudWatch[15], together expose functionalities which are required for undertaking provisioning of application services on EC2. The Elastic Load Balancer service automatically provisions incoming application workload across available EC2 instances while the Auto Scaling service can be used to dynamically scale-in or scale-out the number of EC2 instances for handling changes in service demand patterns. Finally the CloudWatch service can be integrated with the above services for strategic decision making based on collected real-time information.

Eucalyptus is an open source cloud computing platform. It is composed of three controllers. Among the controllers, the *cluster controller* is a key component that supports application service provisioning and load balancing. Each cluster controller is hosted on the *head node* of a cluster to interconnect the outer public networks and inner private networks together. By monitoring the state information of instances in the pool of server controllers, the cluster controller can select any available service/server for provisioning incoming requests. However, as compared to Amazon services, Eucalyptus still lacks some of the critical functionalities, such as auto scaling for its built-in provisioner.

Fundamentally, Microsoft Windows Azure *fabric* has a weave-like structure, which is composed of node including servers and load balancers, and edges including power

[12]`http://www.eucalyptus.com/`

[13]`http://aws.amazon.com/elasticloadbalancing/`

[14]`http://aws.amazon.com/autoscaling/`

[15]`http://aws.amazon.com/cloudwatch/`

and Ethernet. The *fabric controller* manages a *service node* through a built-in service, named Azure Fabric Controller Agent, running in the background, tracking the state of the server, and reporting these metrics to the controller. If a fault state is reported, the controller can manage a reboot of the server or a migration of services from the current server to other healthy servers. Moreover, the controller also supports service provisioning by matching the VMs that meet required demands.

GoGrid Cloud Hosting offers developers the F5 Load Balancer[16] for distributing application service traffic across servers, as long as IPs and specific ports of these servers are attached. The load balancer provides the round robin algorithm and least connect algorithm for routing application service requests. Additionally, the load balancer is able to detect the occurrence of a server crash, redirecting further requests to other available servers. But currently, GoGrid only gives developers a programmatic set of APIs to implement their custom auto-scaling service.

Unlike other cloud platforms, Google App Engine offers developers a scalable platform in which applications can run, rather than providing direct access to a customized virtual machine. Therefore, access to the underlying operating system is restricted in App Engine where load-balancing strategies, service provisioning, and auto scaling are all automatically managed by the system behind the scenes. Thus the implementation is largely unknown. But based on the results in Section 3.3, its provisioning is equipped with a rule-based filtering feature for security reasons, such as denial of service attacks.

In addition, Chohan et al. (2009) have presented initial efforts of building App Engine-like framework, *AppScale*, on top of Amazon EC2 and Eucalyptus. Their offering consists of multiple components that automate deployment, management, scaling, and fault tolerance of an App Engine application. In their design and implementation, a single *AppLoadBalancer* exists in AppScale for distributing initial requests of users

[16]`http://www.gogrid.com/cloud-hosting/load-balancers.php`

to the *AppServer*s of App Engine applications. The users initially contact AppLoader-Balancer to request a login to an App Engine application. The AppLoadBalander then authenticates the login and redirects request to a randomly selected AppServer. Once the request is redirected, the user can start contact the AppServer directly without going through the AppLoaderBalancer during the current session. The *AppController* sit inside the AppLoadBalancer is also in charge of monitoring the AppServers for growing and shrinking as the AppScale deployments happen over the time.

There is no single cloud infrastructure provider has their data centers at all possible locations throughout the world. As a result, all cloud application providers currently have difficulty in meeting SLA expectations for all their customers. Hence, it is logical that each would build bespoke SLA management tools to provide better support for their specific needs. This kind of requirements often arises in enterprises with global operations and applications such as Internet service, media hosting, and Web 2.0 applications. This necessitates building technologies and algorithms for seamless integration of cloud infrastructure service providers for provisioning of services across different cloud providers.

2.3 Overview of cloud databases

Over the past decade, rapidly growing Internet-based services have substantially redefined the way of data persistence and retrieval. End-users can not only easily consume content provided, but also provide content any form, with the recent advances in the web technology. For example, building a personal web page with Google Sites[17], starting a blog with WordPress[18], Blogger[19], or LiveJournal[20], and making both publicly search-

[17] `http://sites.google.com/`

[18] `http://wordpress.org/`

[19] `http://www.blogger.com/`

[20] `http://www.livejournal.com/`

able for end-users all over the world have now become a commodity. Arguably, the main goal of the next wave is to facilitate the job of implementing every application as a distributed, scalable, and widely-accessible service on the web. Services such as Facebook[21], Flickr[22], YouTube[23], Zoho[24], and Linkedin[25] are currently leading this approach. Such applications are both data-intensive and very interactive. For example, the Facebook social network contains 500 million end-users[26]. Each end-user has an average of 130 friendship relations. Moreover, there are about 900 million objects with which registered end-users interact such as: pages, groups, events, and community pages. Other smaller scale social networks such as LinkedIn, which is mainly used by professionals has more than 80 million registered end-users. Therefore, it becomes an ultimate goal to make it easy for everybody to achieve such high scalability and availability goals with minimum effort.

In general, relational database management systems (RDBMSs), namely MySQL, PostgreSQL, SQL Server, and Oracle, have been considered as the one-size-fits-all solution for data persistence and retrieval for decades. They have matured after extensive research and development efforts, and have very successfully created a large market and solutions in different business domains. However, ever-increasing needs for scalability and new application requirements have created new challenges, thus, leading to some dissatisfaction with this one-size-fits-all approach in some web scale applications (Stonebraker et al., 2007; Stonebraker and Cetintemel, 2005).

2.3.1 Related concepts in cloud databases

[21]`http://www.facebook.com/`

[22]`http://www.flickr.com/`

[23]`http://www.youtube.com/`

[24]`http://www.zoho.com/`

[25]`http://www.linkedin.com/`

[26]`http://www.facebook.com/press/info.php?statistics`

Scaling up vs. scaling out

Nowadays, building web applications is commonly based on a three-tier approach, including the web server layer, the application server layer, and the data server layer. In practice, when the application load increases, there are two main options for achieving scalability at each tier and enable the application to be able to cope with more requests, as illustrated in Figure 2.2:

1. *Scaling up*: aims at allocating a bigger machine with more horsepower, such as more processors, memory, and bandwidth, to handle increasing application loads. It is also known as vertical scalability.

2. *Scaling out*: aims at replicating the service layer across more machines so that newly added replicas can handle increasing requests. It is also known as horizontal scalability.

The scaling up option has the main drawback that large machines are often very expensive and eventually a physical limit is reached where a more powerful machine cannot be purchased at any cost. Alternatively, scaling out by replication is a well-known strategy to achieve the availability, scalability, and performance improvement goals in the distributed system (Kemme et al., 2010). It is both extensible and economical - especially in a dynamic workload environment - to scale out by adding storage space or buying another commodity server, which fits well with the new pay-per-use philosophy of cloud computing.

In general, the web server layer and the application server layer are easy to scale out, because any new replicas of these services can operate completely independently of other replicas. In contrast, the data server layer has a limited ability to scale out, as RDBMS is a stateful design which needs to guarantee a consistent view of the system for requests of the service.

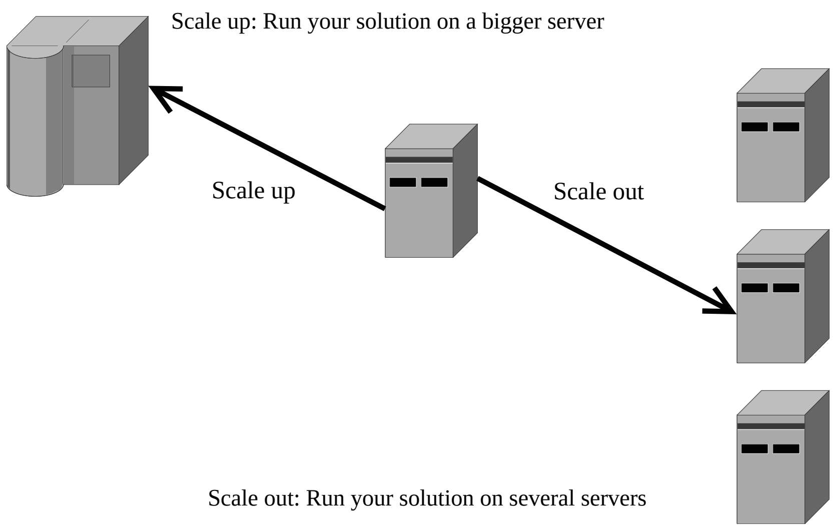

Scale up: Run your solution on a bigger server
Scale up
Scale out
Scale out: Run your solution on several servers

Figure 2.2: Scaling up vs. Scaling out

CAP theorem

The *CAP* theorem shows that a shared-data system can only choose at most two out of three properties (Brewer, 2000, 2012; Gilbert and Lynch, 2002, 2012): *Consistency* where all records are the same in all replicas, *Availability* where all replicas can accept updates or inserts, and tolerance to *Partitions* where the system still functions when distributed replicas cannot talk to each other.

In practice, it is highly important for cloud-based applications to be always available to accept update requests of data, meanwhile at the same time support non-blocking data updates even while the same data is being read for scalability reasons. Therefore, when data is replicated over a wide area, this essentially just leaves a system with only one possible selection between consistency or availability. Thus, the consistency part is typically compromised to yield reasonable system availability (Abadi, 2009, 2012). Hence, most of the cloud database management solutions overcome the difficulties of distributed replication by relaxing the consistency guarantees of the system. Indeed, they implement various forms of weaker consistency models, such as eventual consistency (Vogels, 2009), timeline consistency, and session consistency (Tanenbaum and Steen, 2006), so that all replicas do not have to agree on the same value of a data item at every moment of time. The eventual consistency policy guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. If no failures occur, the maximum size of the inconsistency window can be determined based on factors such as communication delays, the load on the system and the number of replicas involved in the replication scheme.

NoSQL movement

While traditional transactional database management applications, such as banking and stock trading, usually tend to rely on strong consistency guarantees and require microsecond precision for their read operations, the eventual consistency model is more favorable to the new generation of many Web 2.0 applications, which could be more tolerant with a wider window of data inconsistency. In practice, several very large Web-based systems such as Amazon[27], Google[28], and Yahoo[29] have relied on the database system that implements eventual consistency model for managing their replicated data over distributed data centers. Such a new generation of database software with low-cost and high-performance has emerged to challenge the dominance of relational database management systems. An important reason for this movement, named as *NoSQL* (*N*ot *O*nly *SQL*), is that database requirements of web, enterprise, and cloud computing applications may vary because of different implementations. Strong data consistency is no longer a necessity for all applications, for many high-volume Web 2.0 applications, such as eBay[30], Amazon, Twitter[31], and Facebook, scalability and high availability are essential requirements that can not be compromised. For these applications, even the slightest

[27]`http://www.amazon.com/`
[28]`http://www.google.com/`
[29]`http://www.yahoo.com/`
[30]`http://www.ebay.com/`
[31]`http://www.twitter.com/`

breakdown can cause significant financial consequences and affect customer trust. In particular, these new NoSQL database systems share a number of common design features (Cattell, 2011), such as scaling out over many servers, simple interface or protocol in contrast to a SQL binding, weak consistency model instead of ACID transactions, distributed indexes and RAM, and semi-structured data schema.

2.3.2 Database management trade-offs

An important issue in designing large scalable database management applications is to avoid the mistake of trying to be “everything for everyone”. Because different systems make various trade-offs to optimize for different purposes, there is no single system that can best suit all kinds of workloads. Therefore, the most challenging aspects in these applications are to identify the most important features of the target application domain and to decide about the various design trade-offs which immediately lead to performance trade-offs. To tackle this problem, Jim Gray (in Hey et al., 2009) came up with the heuristic rule of ”20 queries”. The main idea of this heuristic is that on each project, we need to identify the 20 most important questions the user wanted the data system to answer. He said that five questions are not enough to see a broader pattern and a hundred questions would result in a shortage of focus.

As a summary, Table 2.3 compares the design decisions of surveyed database management systems from the following two subsections, Section 2.4 and 2.5, where the details of each system are presented. In general, it is difficult to guarantee ACID properties for replicated data over large geographic distances. Cooper et al. (2010) discussed the trade-offs facing cloud database management applications as follows:

- *Read performance versus write performance*: An update to a record can either attach the delta to the existing record, or completely overwrite the existing one. The former is write-efficient, as the write costs are limited to only that of modified

Table 2.3: Design decisions of various cloud database management systems

System	Data Model	Query	Consistency	CAP	License
Dynamo	Key-Value	API	Eventual	AP	Inter@AMZN
PNUTS	Key-Value	API	Timeline	AP	Inter@YHOO
Bigtable	Col. Families	API	Strict	CP	Inter@GOOG
Cassandra	Col. Families	API	Tunable	AP	Apache
HBase	Col. Families	API	Strict	CP	Apache
Hypertable	Mul-dim. Tab	API/HQL	Eventual	AP	GNU
CouchDB	Document	API	Eventual	AP	Apache
SimpleDB	Key-Value	API	Multiple	AP	Commercial
S3	Large Obj.	API	Eventual	AP	Commercial
Table Storage	Key-Value	API/LINQ	Strict	AP/CP	Commercial
Blob Storage	Larg Obj.	API	Strict	AP/CP	Commercial
Datastore	Col. Families	API/GQL	Strict	CP	Commercial
RDS	Relational	SQL	Strict	CA	Commercial
Azure SQL	Relational	SQL	Strict	CA	Commercial
Cloud SQL	Relational	SQL	Strict	CA	Commercial

bytes. However, in contrast to that of the write operation, the former read operation is inefficient as there is a cost incurred in the reconstruction of deltas.

- *Latency versus durability*: Synchronizing writes immediately to disk before responding success takes a longer time than storing writes in memory and synchronizing later to disk. The latter approach avoids costly disk I/O operations to reduce write latency. However, the unsynchronized data could be lost if system failures happen before the next synchronization.

- *Synchronous versus asynchronous replication*: Synchronous replication keeps all replicas up to date during the time, but potentially incurs high latency on updates.

Furthermore, availability of the system may be affected if synchronization is suspended due to some replicas being offline. Asynchronous replication avoids high write latency over networks but allows stale data. Moreover, data loss may occur if an updated replica goes offline before propagating data.

- *Data partitioning*: Data can be partitioned strictly on row basis or on column basis. Row-based partitioning allows efficient access to an entire record. Hence it is ideal for accessing a few records in their entirety. Column-based storage is more efficient for accessing a subset of the columns, particularly when multiple records are accessed.

Kraska et al. (2009) have argued that finding the right balance between cost, consistency and availability is not a trivial task for designing large scale database management applications. Hence, they presented a mechanism that not only allows designers to define the consistency guarantees based on the data at the transaction level but also allows for the ability to automatically switch consistency guarantees at runtime. They described a dynamic consistency strategy, called *consistency rationing*, to reduce the consistency requirements when possible, for example when the penalty cost is low, and raise them when it matters, for example when the penalty costs would be too high. The adaptation is driven by a cost model and different strategies that dictate how the system should behave. In particular, they divide the data items into three categories (A, B, C) and treat each category differently depending on the consistency options provided. The A category represents data items for which we need to ensure strong consistency guarantees as any consistency violation would result in large penalty costs, the C category represents data items that can be treated using session consistency as temporary inconsistency is acceptable while the B category comprises all the data items where the consistency requirements vary over time depending on the actual availability of an item. Therefore, the data of this category is handled with either strong or session consistency depending on a statistical-based policy for decision making.

LazyBase, proposed by Keeton et al. (2010), is another approach for managing data freshness and query performance trade-offs in large scalable database management applications. Instead of switching consistency guarantees automatically, it accepts users' specifications of query freshness and performance goals. It is designed to break metadata processing into a pipeline of ingestion, transformation, and query stages. Each stage can be scheduled independently for a given set of metadata, therefore allowing stage processing in parallel for high performance and efficiency. It also allows different stages of the pipeline to be queried independently, thus avoiding possible freshness delay. Cipar et al. (2012) later improved LazyBase with focus on batching approach, fault tolerance, and scaling. While consistency rationing and LazyBase represent two approaches that adaptive consistency management for cloud databases, but they mainly target the perspective of cloud providers. There is also a lack of a general reusable infrastructure and customizable components that could integrate customer-centric performance monitoring data.

There have been efforts in designing scalable database management applications with transactional supports. Das et al. (2009) have proposed *ElasTraS*, which is an elastic and scalable transactional data store. However, it can only provide limited transactional semantics to a single partition. Such a design is expected to workloads that are limited to single object accesses. Later, in the study of *G-Store*, Das et al. (2010) made transactions available and fault-tolerant across multiple horizontal partitions with the idea of *key group protocol*. However, transactions are only allowed within a dynamic group, not allowed across these formed groups, because a given key can only participate a single group at any time. This design is expected to work in scenarios of exclusive collaborations, such as online games. Sovran et al. (2011) have used *parallel snapshot isolation* in *Walter* to relief pain of write-write transaction conflicts in developers' minds. The parallel snapshot isolation allows optimized transactions to execute within a single site, but enforces satisfactory of causal relationships of keys before requiring a two-phase commit across data centers. In contrast, *COPS* by Lloyd et al. (2011) bears conceptual similarity to ensure casual dependencies between keys, but with the focus of availability and low-latency. Corbett et al. (2012) described Google *Spanner* of being capable of enabling global commit timestamps to transactions. It is achieved with new *TrueTime* API. However, the details of the API haven't been released yet. And finally, (Thomson et al., 2012) proposed *Calvin* as a generic scalable transactional layer which is expected to work with any non-transactional, unreplicated data stores.

The transactional supports discussed above are mainly focus on NoSQL database management systems. Nevertheless, many interesting researches have also been conducted within the scope of RDBMSs. Lomet et al. (2009); Lomet and Mokbel (2009) proposed a radically different approach towards large scalable database management applications in the cloud by "unbundling" the database. A database engine is refactored into two layers, the *transaction component* and the *data component*. The transaction component has little idea about the physical data location. But it can control a lock manager and a log manager to impose concurrent control and undo/redo recovery in logical level, guaranteeing no conflict concurrent operations to the data component. The data component, in contrast, only knowns storage structure for indexing, caching, and disk management. It should only concern about the atomic record operations, instead of transaction properties which is managed by the transaction component. The design is implemented and demonstrated in *Deuteronomy* by Levandoski et al. (2011). Moreover, (Thomson et al., 2012) claimed to bear a similar concept in building Calvin. The idea of *scheduling layer* can be mapped to the transaction component. While the idea of *storage layer* can be mapped to the data component. And the *sequencing layer* is further separated for handling data replication.

Florescu and Kossmann (2009) argued that in cloud environments, the main metric that needs to be optimized is the cost as measured in dollars. Therefore, the big challenge of database management applications is to be able determine the right number of machines to meet the performance requirements of a particular workload under an acceptable cost. Hence, performance requirements such as how fast a database workload can be executed or whether a particular throughput can be achieved is no longer the main metric any more. This argument fits well with the rule of thumb calculation which has been proposed by Gray (2008) regarding the opportunity costs of distributed computing in the Internet as opposed to local computations. Gray argues that for outsourcing computing tasks, network traffic fees may outnumber the savings in processing power. In principle, it is useful to take into account the economic factors in formulating the trade-off calculation between basic computing services. This method can easily be applied to the pricing schemes of cloud computing providers, such as Amazon, Google, and Microsoft. Florescu and Kossmann (2009) have also argued in the new large scale web applications, the requirement of providing full read and write availability for all users has surpassed the importance of the ACID paradigm in data consistency. In this circumstance, blocking any valid user request is never allowed. Therefore, in order to minimize the cost of resolving inconsistencies, it is better to design a system that deals with resolving inconsistencies rather than having a system that prevents inconsistencies under all circumstances.

2.4 NoSQL database systems: state-of-the-art

2.4.1 Key database management systems

This subsection provides an overview of the main NoSQL systems, which has been introduced and internally used by three of the big players in the scalable database management domain: Amazon, Yahoo, and Google.

Amazon Dynamo

As a high-volume web site, reliability is essential to Amazon because even the slightest downtime can cause significant financial consequences and affect customer trust. Amazon *Dynamo* originates from Amazon, aiming to serve tens of millions of customers with tens of thousands of servers that are geographically distributed over the world.

The Amazon Dynamo system is a highly available and scalable distributed key-value based datastore implemented for internal Amazon applications (DeCandia et al., 2007). The design of Dynamo system is based on two concerns of using a relational database. On the one hand, although the relational database can provide complex data schema, in practice, many applications in Amazon only require simple primary key access. Thus, the query model of the Dynamo system is key-based with single read and write operations where there is no operation that spans multiple data items. On the other hand, a relational database tends to be limited in scalability and availability according to common patterns. However, the Dynamo system implements an innovative *Dynamo ring* to enhance replications.

In order to distribute workload across multiple hosts, Amazon Dynamo uses a variant of the consistent hashing mechanism (Karger et al., 1997) for partitioning. This mechanism defines a fixed circular space or ring first as the output range of a hash function. Then, a random value in the range of the space is assigned to each node, known as the *position* of the node on the ring. Hence, each data item is stored in a node position that is the closest in the clockwise direction to the other data item's position as determined by hashing the item's key. Thus, each node is only in charge of the range of the ring from it to its previous node, while adding or removing a node on the ring have no impact on other nodes except its neighbors.

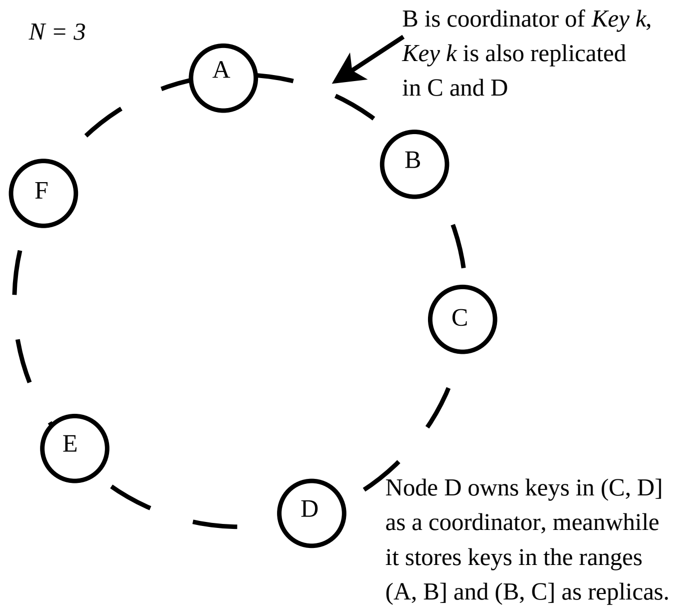

N = 3
B is coordinator of Key k,
Key k is also replicated
in C and D
A
B
C
D
E
F
Node D owns keys in (C, D]
as a coordinator, meanwhile
it stores keys in the ranges
(A, B] and (B, C] as replicas.

Figure 2.3: Partitioning and replication of data in Dynamo ring (DeCandia et al., 2007)

In the Amazon Dynamo system, each data item identified by a key k is assigned to a *coordinator* and $N - 1$ clockwise successor nodes for replication where N is a configurable parameter. The coordinator owns the data items that fall within its range, and takes care of the responsibility of the replication of them. As a result, each node stores data items in the range of the ring from it to its N^{th} predecessor. As illustrated in Figure 2.3, node B owns a copy of the data with key k locally, as well as replicates it at nodes C and D. Node D stores the data with keys within the ranges $(A, B]$, $(B, C]$, and $(C, D]$, and takes care of the data with keys that fall in the range of $(C, D]$.

Yahoo! PNUTS

Yahoo! *PNUTS* system, lately renamed to Sherpa, is a scalable database system, storing tables of records with attributes to support web applications internally in Yahoo! (Cooper et al., 2008). The main goal of the system is serving data. Therefore, a list of functions is enhanced to support this goal. Firstly, a simple relational model is supported, avoiding complex queries. Secondly, *blob* is validated as a main data type, storing arbitrary structures in a record, in addition to large binary objects like image or audio. Thirdly, the data schema of tables is enforced in a flexible way, allowing the ability to add attributes at any time and also keep values of attributes empty in a record.

Figure 2.4 illustrates the system components of Yahoo! PNUTS. A region is a basic unit, which contains complement system components such as storage units, tablets, tablet controllers, and routers, as well as a full copy of tables. In practice, the PNUTS system consists of multiple geographically distributed regions. On the physical level, *tablets* that are horizontal partitions of data tables are scattered across storage units in many servers. In each server, the number of tablets is variable, due to workloads balancing, which shifts tablets from overloaded servers to underloaded ones. Hence, hundreds to thousands of tablets can be achieved in a server. The *router* can determine the location of a given record in two steps. First, it resolves which tablet has a given record by querying the cached interval mapping, which defines tablet boundaries, and maintains mapping correlations of tablets and storage units. Then, it determines which storage unit owns a given tablet, by applying mapping correlations to the given tablet. The *tablet controller* is the owner of interval mapping. It is also in charge of tablet management, such as moving a tablet across storage units for workload balancing or data recover, or splitting a large tablet.

As mentioned above, the system is designed for serving data that consists mainly of queries of single record or small groups of records. The query model is designed with simplicity in mind. Thus, it provides selection and projection options of a single table, but no join operation as it is too expensive to provide. It also allows updating and deleting operations only on primary key basis. Moreover, for reading multiple records, it supports a *multiget* operation for retrieving data in parallel.

Yahoo! PNUTS provides a consistency model that supports a variety of levels between that of general serializability to eventual consistency (Vogels, 2009). The model

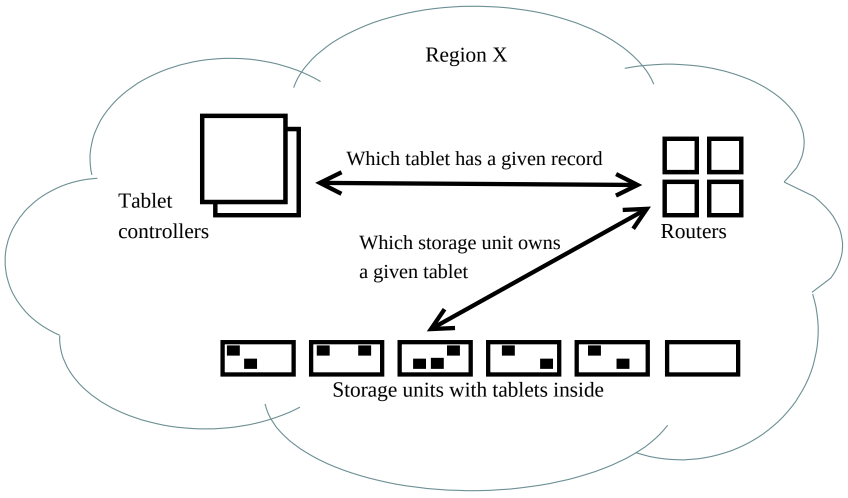

Region X
Which tablet has a given record
Tablet controllers
Routers
Which storage unit owns a given tablet
Storage units with tablets inside

Figure 2.4: PNUTS system components (Cooper et al., 2008)

is developed based on the realization that web applications normally operate one record at a time whereas different records may be manipulated in different geographic areas. Thus, the model defines *per-record timeline* consistency for a given record where all updates to the record are applied in the same order across replicas. Specifically, for each record, if one replica receives the most write for a specific record, the replica is elected as the master that maintains the updated timeline of the record. The per-record timeline consistency model can be divided into various levels of consistency guarantees (Cooper et al., 2008), including:

- *Read-any*: Read any version of the record where it is possible to return a stale version.
- *Read-critical (required version)*: Read a version of the record that is newer than, or the same as the *required version*.
- *Read-latest*: Read the latest version of the record for all successful write.
- *Write*: Write a record without reading its value in advance. This may result in blind writes.

- *Test-and-set-write (required version)*: Write a record if and only if the current version is equivalent to the requirement version. It can be used as an incremental counter.

Google Bigtable

Google *Bigtable* is used as a scalable, distributed storage system (Chang et al., 2008) in Google for a great number of Google products and projects such as: Google Docs[32], Google Earth[33], Google Finance[34], Google search engine[35], and Orkut[36]. These products can configure Bigtable for a variety of usages, supporting workloads from throughput-oriented job processing to serving latency-sensitive data, spanning servers from a handful number to thousands of commodity servers, and scaling data from a few bytes to a size of petabytes.

The data model designed in Bigtable is not a relational data model, but a simple data model with dynamic control. Thus, end-users can change the data layout and data format without being restricted by data schemas. In particular, Bigtable uses a sparse, multidimensional, sorted map to store data. Each cell in the map can be located by a row key, a column name, and a timestamp. A concrete example that reflects some of the main design decisions of Bigtable is the scenario of storing a collection of web pages. Figure 2.5 illustrates an example of this scenario where URLs are used as row keys and various web elements as column names. Values of web elements such as contents and anchors of the web page are in versioned cells under the timestamps when they were fetched.

The row keys are sorted in lexicographic order in Google Bigtable. Every single row

[32]http://docs.google.com/
[33]http://earth.google.com/
[34]http://www.google.com/finance
[35]http://www.google.com/
[36]http://www.orkut.com/

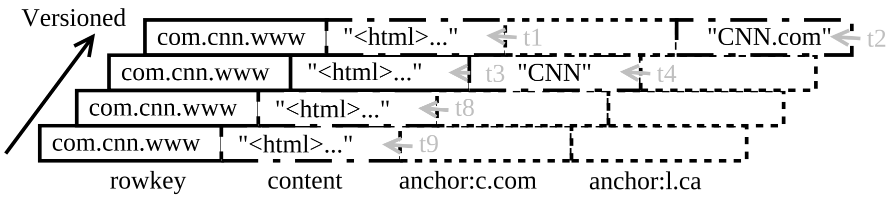


Figure 2.5: Sample Bigtable structure (Chang et al., 2008)

key is an atomic unit of a read or write operation. Usually, ranges of row keys, named *tablets*, can dynamically span multiple partitions for distribution and load balancing. Therefore, a table with multiple ranges can be processed in parallel on a number of servers. Each row can have an unlimited number of columns. Sets of them are grouped into *column families* for access control rights. Each cell is versioned and indexed by timestamps. The number of n versions of a cell can be declared, so that only recent n versions are kept in decreasing timestamp order.

The Google Bigtable provides low-level APIs for the following functions: creating, deleting, and changing tables and column families; updating configurations of cluster and column family metadata; adding, removing, and searching values from individual rows or a range of rows in a table. However, Bigtable does not support general transactions across row keys. Only atomic read-modify-write sequences on a single row, known as *single-row* transactions, are allowed.

On the physical level, the distributed Google File System (GFS), introduced by Ghemawat et al. (2003), is used to store Google Bigtable log and data files. The data is in Google *SSTable* file format, which offers an ordered, immutable keys to values map for persistence. Bigtable relies on a distributed lock service called *Chubby* (Burrows, 2006) which uses the Paxos algorithm (Chandra et al., 2007) to keep itself fault-tolerant if a majority of five composing replicas are accessible to each other. Among the five replicas, one of them is voted as *master*, proactively serving all requests and balancing workloads across tablet servers. Each Bigtable has to be allocated to one master server and a number of tablet servers to be available. Hence, Bigtable can not work properly without Chubby as it is necessary for keeping the master server running and for storing information of Bigtable, such as bootstrap locations, schemas, and access control lists.

It is worth mentioning that successors have been created in the last few years after the first release of Google Bigtable. The first successor is Google *Megastore*, which comes with semi-relational data model and support for synchronous replication. The second successor is Google Spanner, which is a scalabing, multi-version, globally-distributed, and synchronously-replicated database. It is used internally for serving $F1$ (Shute et al., 2012).

2.4.2 Open source projects

Most NoSQL key database management systems, such as Google Bigtable, Yahoo! PNUTS, and Amazon Dynamo, are for internal use only and are not available for cloud customers. Therefore, many open source projects have been built to implement the concepts of these key systems and make it available for cloud customers. These systems attract a lot of interest from the research and industry community, and eventually they have evolved into various systems. In this subsection, only some of these projects will be introduced briefly.For the full list of NoSQL databases, the NoSQL database website[37] provides an up-to-date list of all NoSQL database systems. Thus far, published details about the implementation of most of these systems have been scarce. However, in general, the NoSQL open source project can be broadly classified into the following categories:

- *Key-value stores*: These systems use the simplest data model which is a collection of objects where each object has a unique key and a a set of attribute/value pairs.

[37]`http://NoSQL-database.org/`

- *Extensible record stores*: They provide variable-width tables (Column Families) that can be partitioned vertically and horizontally across multiple nodes.
- *Document stores*: Where the data model consists of objects with a variable number of attributes with a possibility of having nested objects.

Cassandra

Cassandra[38] is known as a highly scalable, eventually consistent, distributed, structured key-value store (Lakshman and Malik, 2010). It is initially designed as an inbox storage service in Facebook and it has been open-sourced since 2008. One of its authors is also an author of Amazon's Dynamo. Hence, Cassandra combines the distribution technology from Amazon Dynamo with the data model from Google Bigtable. This results in a system where comes which combines the Dynamo's eventual consistent feature with Bigtable's column family-based data model.

The data model comes with four basic concepts. The basic unit of the data model is the *column* which includes a name, a value and a timestamp. A *column family* groups multiple columns together, comparable with the table of a relational database. Column families can be composed into a *keyspace*, which can be considered as a schema to a relational database, typically, one keyspace is used per application. *Super columns* represent columns that themselves have subcolumns, such as maps.

Cassandra offers various levels of consistency models that are suitable for specific applications. In particular, for every read and write operation, there are seven and eight consistency options available respectively, in version 1.2.

HBase

HBase[39] is another project based on the ideas of Google's Bigtable system. It builds

[38]`http://cassandra.apache.org/`

[39]`http://hbase.apache.org/`

on top of the Hadoop Distributed File System (HDFS)[40] as its data storage engine. The advantage of this approach is that HBase does not need to worry about data replication, data consistency, and resiliency because HDFS already provides them. However, the downside is that it inherits the limitations of HDFS, which is that it is not optimized for random read access.

In the HBase architecture, data is stored in a farm of Region Servers. A *key-to-server* mapping is used to locate the corresponding server. The in-memory data storage is implemented using a distributed memory object caching system called *Memcache*[41] while the on-disk data storage is implemented as a HDFS file residing in a Hadoop data node server.

Hypertable

The *Hypertable*[42] project is designed to achieve a high performance, scalable, distributed storage and processing system for structured and unstructured data. As with HBase, Hypertable also runs on top of HDFS that offers automatic data replication, data consistency and resiliency. Consequently, Hypertable also suffers from the same limitations of inefficient random data access.

In Hypertable, the data model is represented as multi-dimensional tables. The system supports create, modify, and query data via low-level APIs or Hypertable Query Language (HQL). Data processing can be executed in parallel to increase performance.

CouchDB

CouchDB[43] is a document-oriented database. A document object, identified by a unique identity, is the primary data unit consisting of named fields and typed field values such

[40]http://hadoop.apache.org/hdfs/

[41]http://memcached.org/

[42]http://hypertable.org/

[43]http://couchdb.apache.org/

as strings, numbers, dates, or even ordered lists and associative maps. Data query is via RESTful HTTP API that offers read, update, add, and delete operations. The system is lockless and optimistic, and there is no partially edited documents saved in system. If two clients try to save the same document, an edit conflict error happens to one client on updating. The system resolves the conflict by reopening the latest document version and reapplying all updates. The document update can either be all, for succeeding entirely, or none, for failing completely.

Other projects

Many other variant projects are recently started to follow the NoSQL movement and support different types of data stores, namely, *Voldemort*[44] and *Dynomite*[45] for key-value stores, *MongoDB*[46] and *Riak*[47] for document stores, and *Neo4j*[48] and *DEX*[49] for graph stores.

2.5 Public cloud databases: state-of-the-art

2.5.1 NoSQL database as a service

NoSQL database as a service is part of the offering of *database as a service*. In general, database as a service is an emerging paradigm for database management in which a cloud service provider hosts a database as a service (Agrawal et al., 2009; Hacigümüs et al., 2002). The service providers charge customers on pay-per-use basis and in return for offering hardware and software, managing system and software upgrades, and maintaining

[44] `http://project-voldemort.com/`
[45] `http://wiki.github.com/cliffmoon/dynomite/dynomite-framework`
[46] `http://www.mongodb.org/`
[47] `http://wiki.basho.com/display/RIAK/Riak`
[48] `http://neo4j.org/`
[49] `http://www.dama.upc.edu/technology-transfer/dex`

administrative and maintenance tasks. It is an attractive solution for various purposes such as for data archiving, development and test, and startup companies, especially as the service comes with the promise reliable, scalable and elastic data storage.

Specifically, NoSQL database as a service uses NoSQL database systems as the back-end of the database systems. Key players, like Google, Amazon, and Microsoft, all provide their own NoSQL database as a service solutions to their customers. It must be noted that based on the cloud providers' offerings there has been projects that federate the existing solutions of NoSQL as a service as a unified storage system and make decisions of selecting the best NoSQL database as a service based on several factors, for example, *MetaStore* makes trade-offs of consistency and latency (Bermbach et al., 2011), while *MetaCDN*[50] is designed for high performance and low cost content delivery (Broberg et al., 2009).

Amazon SimpleDB

Generally, Amazon SimpleDB is designed for running queries on structured data. In SimpleDB, data in is organized into *domains* which is similar to tables, within which we can put data, get data or run queries. Each domain consist of *items* which is equivalent to records, described by pairs of *attribute* names and values. It is not necessary to pre-define all of the schema information as new attributes can be added to the stored dataset when needed. Thus, the approach is similar to that of a spreadsheet and does not follow the traditional relational model. SimpleDB provides a small group of API calls that enables the core functionality to build client applications such as: *CreateDomain*, *DeleteDomain*, *PutAttributes*, *DeleteAttributes*, *GetAttributes* and *Select*. The main focus of SimpleDB is to provide fast reading. Therefore, query operations are designed to run on a single domain. SimpleDB keeps multiple copies of each domain where a successful write operation guarantees that all copies of the domain will durably persist. In

[50]`http://www.metacdn.com/`

particular, SimpleDB supports two read consistency options: eventually consistent read and consistent read.

Amazon S3

Similar to Amazon SimpleDB, Amazon has not published the details of its other product, Amazon Simple Storage Service (S3). Conceptually, S3 is an infinite store for objects of variable sizes. Each object is a container of bytes. It is identified by a URI. With the specified URI, clients are able to access via SOAP or REST-based interfaces remotely, for example, API *GETS* returns an object and API *PUTS* writes a new version of the object. Ideally, S3 can be considered as an online backup solution or for archiving large objects, which are not frequently updated. S3 provides read-after-write consistency for *PUTS* of new objects and eventual consistency for overwrite *PUTS* and *DELETES*.

In the study conducted in Chapter 4, there is no stale data observed for overwrite *PUTS* operations. The study was conducted with several purposes. One of them was to investigate possible eventual consistency models that NoSQL database as a services could offer. Among all eventual consistency models (Vogels, 2009), some are sensitive to the locations of writer and reader processes, such as read-your-writes consistency. In order to detect these models, the study implemented one writer thread and one or multiple reader threads with five configurations, including in the same thread, in different threads, in the same region but different instances, and in different regions and instances. Nevertheless, multiple reader threads were never distributed, because synchronizing time across multiple instances is not trivial in cloud. As shown in Section 5.1.3, the time differences could be as greater as 40 milliseconds within a duration of 20 minutes. Furthermore, it could be tens of seconds differences after 24 hours with simply a standard time synchronization configuration. An implementation of distributed reader threads is conducted in (Bermbach and Tai, 2011)'s study. It is interesting that a contradicted result was observed in their report. It claims the observation of S3's consistency behavior, and categorizes it into a LOW phase and a SAW phase. However, in a recent communication, the first author mentioned that there was no observation of the two phases with the same experiment configuration in a recent re-run. Only small inconsistency, similar to observations in previous LOW phase, could be observed. We would argue that such a small inconsistency window is likely caused by time synchronization. We would also like to confirm that there was no inconsistent data observed in our recent re-run with our setup. Once an update of a S3 object is confirmed, all following reads return the up-to-date value. It seems the contradict results could be caused by the S3 service itself. And possibly, a recent upgrade has improved its performance in distributed read and thus removed such an observation.

Although the implementation of S3 is largely unknown, Brantner et al. (2008) have presented initial efforts of building web-based database applications on top of S3. They described various protocols in order to operate S3 in the same manner of a relational database. In their system, the *record manager* component is designed to create, read, update and scan records where each record contains a key and payload data. The size of a record must be no larger than a page size, as a page is a container of records, and each page is physically stored in S3 as a single object. In addition to record manager, a buffer pool is also implemented in the *page manager* component. The buffer pool interacts with S3 like a normal buffer pool in any standard database system: reading pages from S3, pinning the pages in the buffer pool, updating the pages in the buffer pool, and marking the pages as updated, while the page manager is mainly in charge of commit and abort transactions. Moreover, they also implemented standard B-tree indexes on top of the page manager and basic redo log records. However, there are still many database-specific issues that have not been addressed in their yet. That, for example, strict consistency and transactions mechanisms. Furthermore, as addressed in the paper, more functionalities can be devised: query processing techniques, such as join algorithms and query optimization techniques, and traditional database functionalities, such as bulkloading to a database, creating indexes, and dropping a whole collection.

Microsoft Windows Azure Storage

The implementation of Microsoft Windows Azure Table Storage and Blob Storage are revealed by Calder et al. (2011). The Table Storage and Blob Storage have been claimed to provide all three properties of CAP theorem within a *storage stamp*, which is simply a cluster of a number of racks of storage nodes. The claim is achieved by making choices between consistency and availability at very fine granularity in different stages (Brewer, 2012), in this case, two stages. It fulfills availability and tolerance to partitions in the stage of *stream layer*, and then satisfies consistency and tolerance to partitions in the stage of *partition layer*. It must be noted that such a design philosophy also appears in Google Megastore (Baker et al., 2011), an enhancement of Google's Bigtable.

The stream layer works as the bottom layer in the storage stamp, acting as a distributed file system which stores, distributes, and replicates data across many servers. It operates on data by using the append-only model, specifically existing data that can not be modified but appended. The append-only model is good for keeping snapshots, checking failures, diagnosing errors, and repairing corruptions. Therefore, with the simple append-only model, the stream layer maintains the availability and tolerance to partitions at the cost of extra I/O due to the need for scalable garbage collection for keeping low space overhead.

The partition layer built on top of the stream layer guarantees strong consistency, as well as stores semantics of lower level data from the stream layer. It understands transactions and provides access points to the objects given in the transactions. An *Object Table* is used in the partition layer for storing higher level object constructs that can grow up to several petabytes. Such a massive table can be split into a number of *RangePartitions* which represents an non-overlapping chunk of continuous rows in an Object Table. All RangeParititons is spread across *Partition Servers*. Because there is no two Partition Servers that can serve the same RangePartition at the same time, the Partition Server is able to provide strong consistency and ordering of concurrent transactions for the RangePartition that is serving.

Although a synchronous replication is guaranteed within a storage stamp in the stream layer, replication across storage stamps via partition layers still uses asynchronous replication. On average, recent updates made by the primary storage stamp within 30 seconds could be lost, as the geographically replicated secondary storage stamp may not receive these commits when a disaster happens to the primary storage stamp.

Google App Engine Datastore

Google App Engine *Datastore*[51] is not externally accessible, as it is the scalable schemaless object data storage sitting behind Google App Engine. The data object is called *entities*, composed of a unique identity and a number of *properties* where one property can hold a typed value or refer to other entities. A *kind* is a container of entities, analogous to the table in a relational database. However, entities are schemaless in the same kind where two entities can have different properties or even different types for the same properties.

Google App Engine Datastore provides APIs in Python[52] and Java[53]. For the Python interface, it includes a rich data modeling API and a SQL-like query language called Google Query Language (GQL)[54]. Figure 2.6 depicts the basic syntax of GQL. For the

[51]`http://code.google.com/appengine/docs/python/datastore/`
[52]`http://www.python.org/`
[53]`http://www.java.com/`
[54]`http://code.google.com/appengine/docs/python/datastore/`

Java interface, it supports two API standards for modeling and querying, namely Java Data Objects (JDO)[55] and Java Persistence API (JPA)[56]. An entity can be retrieved with its identity or by querying its properties. A query can return from 0 to a maximum of 1000 sorted-by-property-values results where the limitations are imposed in view of memory and runtime constraints. In principle, join is not supported in the query.

Google App Engine Datastore supports transaction. A transaction ensures that operations in a transaction succeed entirely or fail completely. A single operation of creating, updating or deleting an entity happens in a transaction implicitly. Meanwhile, a group of operations can be explicitly defined as a transaction. The Datastore manages transactions in an optimistic manner. The Datastore replicates data to multiple locations. Among all replicas, one is selected as the primary replica to keep the view of the data consistent by replicating delta data to other locations. In the case of failures, the Datastore can wait for the primary to become available, or continue accessing data from an alternative replica, depending on the selection of read policies: strong consistency means reading from the primary replica, while eventual consistency means reading from an alternate replica when the primary location is unavailable.

```
SELECT [* | __key__] FROM <kind>
[WHERE <condition> [AND <condition> ...]]
[ORDER BY <property> [ASC | DESC] [,<property> [ASC | DESC]...]]
[LIMIT [<offset>,]<count>]
[OFFSET <offset>]

<condition> := <property> {< | <= | > | >= | = | != } <value>
<condition> := <property> IN <list>
<condition> := ANCESTOR IS <entity or key>
```

Figure 2.6: Basic Google Query Language syntax

gqlreference.html

[55]http://code.google.com/appengine/docs/java/datastore/jdo/

[56]http://code.google.com/appengine/docs/java/datastore/jpa/

2.5.2 Relational database as a service

The *relational database as a service* is another approach in which a third party service provider hosts a relational database as a service (Agrawal et al., 2009). Such services alleviate the need for their customers to purchase expensive hardware and software, deal with software upgrades and hire professionals for administrative and maintenance tasks. For example, Amazon Relational Database Service (RDS) provides access to the capabilities of MySQL or Oracle database while Microsoft Windows Azure SQL Database has been built on Microsoft SQL Server technologies. As such, customers of these services can leverage the capabilities of traditional relational database systems such as creating, accessing and manipulating tables, views, indexes, roles, stored procedures, triggers, and functions. It can also execute complex queries and joins across multiple tables. The migration of the database tier of any software application to a relational database service is supposed to require minimal effort if the underlying RDBMSs of the existing software application is compatible with the offered service. However, limitations or restrictions are compelled by the service providers for different reasons, for example, no custom plug-in support. Such limitations and restrictions create barriers for possible experimental explorations. Besides certain relational database systems, many other systems, such as DB2 and PostgreSQL, are not yet supported by the relational database as a service approach.

Amazon RDS

Amazon RDS is a new service, which gives access to the full capabilities of a MySQL database, and lately, extended to a larger range of relational database systems, including Oracle and Microsoft SQL Server. Hence, the code, applications, and tools, which are already designed on existing relational database systems can work seamlessly with RDS. Once the database server is running, RDS can automate common administrative tasks such as performing backups or patching the database software. RDS can also manage the task of scaling resources, synchronizing data replication, and automatic failover management, but most of these functionalities require a restart of the running database server, which therefore may interrupt running services. Furthermore, RDS can also integrate with Amazon CloudWatch to monitor utilization metrics, such as CPU, disk I/O, and memory. However, there have been many limitations to RDS, one of the most important is that there is no support for geographic replication. In RDS, the capacity of each database is in a range from 5 GB to 1024 GB. As suggested by Amazon, a higher allocated storage may be able to improve the input/output operation per second (IOPS) performance, as a larger storage is likely to span across multiple Amazon Elastic Block Store (EBS) volumes.

Microsoft Windows Azure SQL Database

Microsoft has recently released the Microsoft Windows Azure SQL Database system[57]. It is announced as a cloud-based relational database service, which has been built on Microsoft SQL Server technologies. Therefore, applications can almost move whatever available operations in SQL Server to Azure SQL such as creating, accessing, and manipulating tables, views, indexes, roles, stored procedures, triggers, and functions. It can execute complex queries and joins across multiple tables. It also supports Transact-SQL (T-SQL), native ODBC, and ADO.NET data access[58]. The implementation details of this project are revealed by Bernstein et al. (2011). A logical database in Azure SQL is called a *table group*. It could be *keyless* like an ordinary SQL Server database which co-locates all tables, or it also could be *keyed* like a partitioned SQL Server database which partitions all of its tables into *row groups* based on a common column called *partitioning key*. Therefore, a transaction of multiple row groups can not be executed as

[57]`http://www.microsoft.com/windowsazure/sqlazure/`

[58]`http://msdn.microsoft.com/en-us/library/h43ks021(VS.71).aspx`

an ACID transaction. A transaction is only executed on a primary server, but it will be propagated to a secondary server shortly. At its core, It is a parallel database system that uses data partitioning on a shared-nothing architecture. In particular, Azure SQL service can be seen as running an instance of SQL Server in a cloud-hosted server, which is automatically managed by Microsoft instead of running an on-premise managed server. In Azure SQL, the size of each hosted database can not exceed the limit of 50 GB.

Google Cloud SQL

Google Cloud SQL[59] is a MySQL database that lives in Google's cloud environment. Not much details has been published on the implementation of this project. However, it is designed to work with Google App Engine and other Google services for small and medium size applications. Therefore, the capability of Cloud SQL instance is limited to 10 GB only. It offers some automatic administrative tasks, such as scheduling backups, patching management, and replicating databases. Although Cloud SQL may not be competitive as other relational database as a service in regards to the range of capabilities, there have been many highlights to Cloud SQL, one of the most important of which is its ability to support synchronous replication in multiple geographic locations.

2.5.3 Virtualized database servers

NoSQL database as a service and relational database as a service offered by cloud providers both come with their own strengths. Firstly, the customers do not have to trouble themselves with administrative work, as the providers deal with software upgrades and maintenance tasks. Secondly, the cloud providers also implemented automatic replication failover and management. But there are obvious shortcomings as well. Firstly, customers may require extra migration efforts on modifying code and converting data.

[59]`https://developers.google.com/cloud-sql/`

Secondly, customers have limited choices, if customers use PostgreSQL or DB2 as their database, there is no simple alternative for both solutions. And thirdly, customers have no full control on achieving the elasticity and scalability benefits.

Therefore, an approach like *virtualized database servers* is necessary sometimes. Thus, the main research presented in this thesis is focused on this approach. For this approach, customers simply port everything designed for a conventional data center into cloud, including database servers, and run in virtual machines. It is worth mentioning that there is no unique approach of deploying virtualized database servers. Therefore, no specific projects and examples will be discussed in this subsection. The virtualized database servers are considered as being good enough, as long as the deployment meets the application requirements.

With such a deployment, there would be minimum changes to existing application code. The customers have full control in configuring the required elasticity of allocated resources (Cecchet et al., 2011; Soror et al., 2008). And the customers can also build low cost solutions for geographic replication by taking advantage of cloud providers' multiple data centers across continents. However, achieving these goals requires the existence of control components (Sakr et al., 2011) which are responsible for monitoring the system state and taking the corresponding actions, such as allocating more/less computing resources, according to the defined application requirements and strategies. Several approaches have been proposed for building control components which are based on the efficiency of utilization of the allocated resources (Cecchet et al., 2011; Soror et al., 2008). The proposed approach in this thesis focuses on building an SLA-based admission control component as a more practical and customer-centric view for achieving the requirements of their applications.

2.6 SLA management for virtualized database servers

An SLA is a contract between a service provider and its customers. SLAs capture the agreed upon guarantees between a service provider and its customer. They define the characteristics of the provided service including service level objectives (SLOs), such as maximum response times, minimum throughput rates, and data freshness, and define penalties if these objectives are not met by the service provider. In general, SLA management is a common general problem for the different types of software systems which are hosted in cloud environments for different reasons such as the unpredictable and bursty workloads from various users in addition to the performance variability in the underlying cloud resources. In particular, there are three typical parties in the cloud. To keep a consistent terminology through out the rest of the thesis, these parties are defined as follows:

- *Cloud service providers*: They offer the client provisioned and metered computing resources, such as CPU, storage, memory, and network, for rent within flexible time durations. In particular, they include: infrastructure as a service providers and platform as a service providers. The platform as a service providers can be further broken into several subcategories of which database as a service provider is one of them.

- *Cloud customers*: They represent the cloud-hosted software applications that utilize the services of cloud service providers and are financially responsible for their resource consumptions. Most of software as a service providers can be categorized into this party.

- *End-users*: They represent the legitimate users for the services or applications that are offered by cloud customers.

While cloud service providers charge cloud customers for renting computing resources to deploy their applications, cloud customers may or may not charge their end-users for processing their workloads, depending on the customers' business model. In both cases, the cloud customers need to guarantee their users' SLA. Otherwise, penalties are applied, in the form of lost revenue or reputation. For example, Amazon found that every 100 ms of latency costs them 1% in sales and Google found that an extra 500 ms in search page generation time dropped traffic by 20%[60]. In addition, large scale Web applications, such as eBay and Facebook, need to provide high assurances in terms of SLA metrics such as response times and service availability to their end-users. Without such assurances, service providers of these applications stand to lose their end-user base, and hence their revenues.

In practice, resource management and SLA guarantee falls into two layers: the cloud service providers and the cloud customers. In particular, the cloud service provider is responsible for the efficient utilization of the physical resources and guarantee their availability for their customers. The cloud customers are responsible for the efficient utilization of their allocated resources in order to satisfy the SLA of their end-users and achieve their business goals. Therefore, there are two types of service level agreements (SLAs):

- *Cloud infrastructure SLA (I-SLA)*: These SLA are offered by cloud providers to cloud customers to assure the quality levels of their cloud computing resources, including server performance, network speed, resources availability, and storage capacity.

- *Cloud application SLA (A-SLA)*: These guarantees relate to the levels of quality for the software applications which are deployed on a cloud infrastructure. In particular, cloud customers often offer such guarantees to their application's end users

[60] `http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html`

in order to assure the quality of services that are offered such as the application's response time and data freshness.

Figure 2.7 illustrates the relationship between I-SLA and A-SLA in the software stack of cloud-hosted applications. In practice, traditional cloud monitoring technologies, such as Amazon CloudWatch, focus on low-level computing resources. However, translating the SLAs of applications' transactions to the thresholds of utilization for low-level computing resources is a very challenging task and is usually done in an ad-hoc manner due to the complexity and dynamism inherent in the interaction between the different tiers and components of the system. In particular, meeting SLAs which are agreed with end-users by cloud customers' applications using the traditional techniques for resource provisioning is a very challenging task due to many reasons such as:

- *Highly dynamic workload*: An application service can be used by large numbers of end-users and highly variable load spikes in demand can occur depending on the day and the time of year, and the popularity of the application. In addition, the characteristic of workload could vary significantly from one application type to another and possible fluctuations on the workload characteristics which could be of several orders of magnitude on the same business day may occur (Bodik et al., 2010). Therefore, predicting the workload behavior and consequently devising an accurate plan to manage of the computing resource requirements are very challenging tasks.

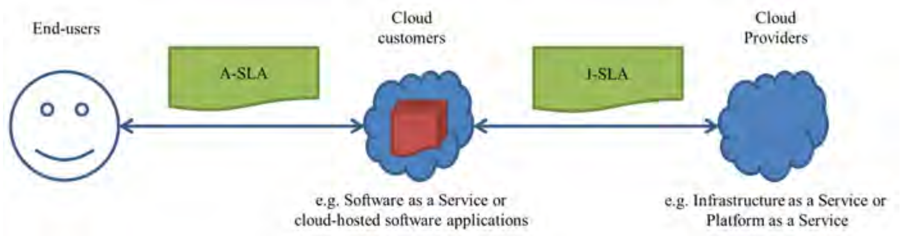

End-users
A-SLA
Cloud customers
I-SLA
Cloud Providers
e.g. Software as a Service or cloud-hosted software applications
e.g. Infrastructure as a Service or Platform as a Service

Figure 2.7: SLA parties in cloud environments

- *Performance variability of cloud resources*: Several studies have reported that the variation of the performance of cloud computing resources is high (Cooper et al., 2010; Lenk et al., 2011; Schad et al., 2010). As a result, currently, cloud service providers do not provide adequate SLAs for their service offerings. Particularly, most providers guarantee only the availability, rather than the performance, of their services (Armbrust et al., 2010; Durkee, 2010).

- *Uncertain behavior*: One complexity that arises with the virtualization technology is that it becomes harder to provide performance guarantees and to reason about a particular application's performance because the performance of an application hosted on a virtual machine becomes a function of applications running in other virtual machines hosted on the same physical machine. In addition, it may be challenging to harness the full performance of the underlying hardware, given the additional layers of indirection in virtualized resource management (Ristenpart et al., 2009).

Several approaches have been proposed for dynamic provisioning of computing resources based on their effective utilization (Cunha et al., 2007; Padala et al., 2007; Wood et al., 2007). These approaches are mainly geared towards the perspective of cloud providers. Wood et al. (2007) have presented an approach for dynamic provisioning of virtual machines. It defines a unique metric based on the data consumption of the three physical computing resources, including CPU, network, and memory to make the provisioning decision. Padala et al. (2007) carried out black-box profiling of the applications and built an approximated model which relates performance attributes such as the response time to the fraction of processor allocated to the virtual machine on which the application is running. Dolly (Cecchet et al., 2011) is a virtual machine cloning technique to spawn database replicas and provisioning shared-nothing replicated databases in the cloud. The technique proposes database provisioning cost models to adapt the provisioning policy to the low-level cloud resources according to application requirements. Rogers et al. (2010) proposed two approaches for managing the resource provisioning challenge for cloud databases. The black-box provisioning uses end-to-end performance results of sample query executions, whereas white-box provisioning uses a finer grained approach that relies on the DBMS optimizer to predict the physical resource consumption, such as disk I/O, memory, and CPU, for each query. Floratou et al. (2011) have studied the performance and associated costs in the relational database as a service environments. The results show that given a range of pricing models and the flexibility of the allocation of resources in cloud-based environments, it is hard for a user to figure out their actual monthly cost upfront. Soror et al. (2008) introduced a virtualization design advisor that uses information about the database workloads to provide offline recommendations of workload-specific virtual machines configurations.

In practice, it is a very challenging goal to delegate the management of the SLA requirements of the customer applications to the cloud service provider due to the wide heterogeneity in the workload characteristics, details and granularity of SLA requirements, and cost management objectives of the very large number of customer applications that can be simultaneously running in a cloud environment. Therefore, it becomes a significant issue for the cloud customers to be able to monitor and adjust the deployment of their systems if they intend to offer viable SLAs to their customers. Failing to achieve these goals will jeopardize the sustainable growth of cloud computing in the future and may result in valuable applications being moved away from the cloud. In the following sections, we present our customer-centric approach for managing the SLA requirements of virtualized database servers.

Chapter 3

A general framework for performance evaluation of cloud platforms

Amazon, Microsoft and Google are investing billions of dollars in building distributed data centers across different continents around the world providing cloud computing resources to their customers. A typical cloud platform includes a cloud application hosting server and a cloud database. Many also offer additional services such as customizable load balancing and monitoring tools. This section focuses on the following three cloud platforms:

- Amazon offers a collection of services, called Amazon Web Services, which includes Amazon Elastic Compute Cloud (EC2) as cloud hosting server, offering infrastructure as a service, Amazon SimpleDB and Simple Storage Service (S3) as cloud databases.
- Microsoft Windows Azure is recognized as a combination of infrastructure as a service and platform as a service. It features *web role* and *worker role* for web hosting tasks and computing tasks, respectively. It also offers a variety of database options including Windows Azure Table Storage and Windows Azure Blob Stor-

age as the NoSQL database options, and Azure SQL Database as the relational database option.

- Google App Engine supports a platform as a service model, supporting programming languages including Python and Java, and Google App Engine Datastore as a Bigtable-based (Chang et al., 2008), non-relational and highly shardable cloud database.

There have been a number of research efforts that specifically evaluate the Amazon cloud platform (Evangelinos and Hill, 2008; Hill and Humphrey, 2009). However, there has been little in-depth evaluation research conducted on other cloud platforms, such as Google App Engine and Microsoft Windows Azure. More importantly, these work lack a more generic evaluation method that enables a fair comparison between various cloud platforms.

In this chapter, a novel approach called CARE (Cloud Architecture Runtime Evaluation) has been developed in an attempt to address the following research questions:

- What are the performance characteristics of different cloud platforms, including cloud hosting servers and cloud databases?
- What availability and reliability characteristics do cloud platforms typically exhibit? What sort of faults and errors may be encountered when services are running on different cloud platforms under high request volume or high stress situations?
- What are some of the reasons behind the faults and errors? What are the architecture internal insights that may be deduced from these observations?
- What are the software engineering challenges that developers and architects could face when using cloud platforms as their production environment for service delivery?

An empirical experiment has been carried out by applying the CARE framework against three different cloud platforms. The result facilitates an in-depth analysis of the major runtime performance differences under various simulated conditions, providing useful information for decision makers on the adoption of different cloud computing technologies.

This chapter presents the CARE evaluation framework in Section 3.1, followed by discussions on the empirical experiment set up and its execution in Section 3.2. Section 3.3 presents the experimental results of all test sets and error analysis captured during the tests. Section 3.4 discusses the application experience of CARE and evaluates the CARE approach.

3.1 The CARE framework

The CARE framework is a performance evaluation approach specifically tailored for evaluating across a range of cloud platform technologies. The CARE framework exhibits the following design principles and features:

- Common and consistent test interfaces across all test targets by employing web services and RESTful APIs. This is to ensure that, as much as possible, commonality across the tests against different platforms is maintained, hence resulting in a fairer comparison.

- Minimal business logic code is placed in the test harness, in order to minimize variations in results caused by business logic code. This is to ensure that performance results can be better attributed to the performance characteristics of the underlying cloud platform as opposed to the test application itself.

- Use of canonical test operations, such as read, write, update, delete. The principle enables simulating a wide range of cloud application workloads using composites of these canonical operations. This approach provides a precise way of describing the application profile.

- Configurable end-user simulation component for producing stepped request volume simulations for evaluating the platform under varying load conditions.
- Reusable test components including test harness, result compilation, and error logging.
- Consistent measurement terminology and metric that can be used across all test case scenarios and against all test cloud platforms.

3.1.1 Measurement terminology

CARE employs a set of measurement terminology that is used across all tests to ensure consistency in the performance instrumentation, analysis and comparison of the results. It considers major variables of interest in the evaluation of cloud platforms, including response time based on those observed by the end-user side, and from the cloud host server side.

Figure 3.1 illustrates the time measurement terminologies in a typical end-user request and round-trip response. From an end-user's perspective, a cloud hosting server and a cloud database provides the following three time-relevant terminologies:

- *Response time* is the total round-trip time, including time taken at the networking layer, as seen by the end-user, starting from sending the request, through to receiving the corresponding response.
- *Processing time* is the amount of time spent on processing the request on the server side.

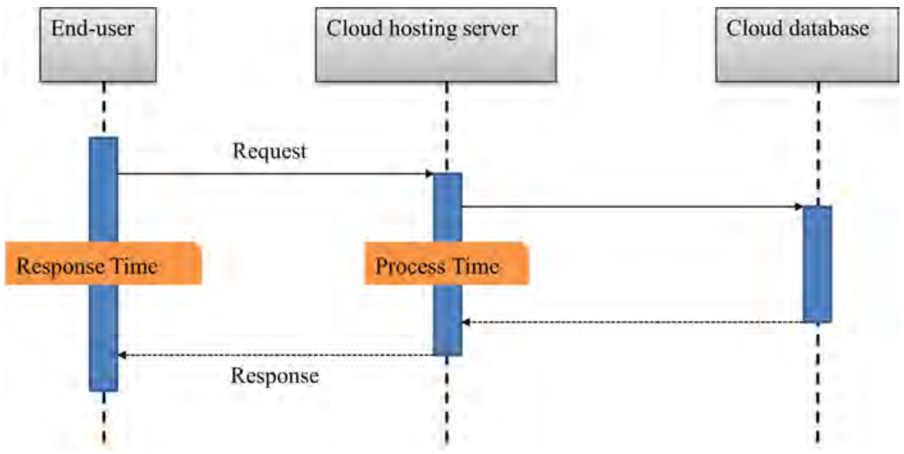


Figure 3.1: Time measurement terminologies

- *Database processing time* is the amount of time a cloud database takes to process a database request. However, it is practically impossible to measure accurately, due to the absence of a timer process in the cloud database. The CARE framework thus equates this measurement to time taken to process the database request as seen by the cloud hosting server by measuring the processing time of the database API as the database processing time as the latency between the hosting servers and cloud databases within the same cloud platform is negligible.

Additional terminologies used refer to different response types that are based on the request:

- *Incomplete request* is a type of request where an end-user fails to send or receive.
- *Completed request* refers to a request where an end-user successfully sends and receives a confirmation response from the cloud platform at completion time.

Subsequently, depending on the response, the completed request can be further classified as:

- *Failed request* that contains an error message in the response.
- *Successful request* which completes the transaction without an error.

3.1.2 Test scenarios

The CARE framework provides three key test scenarios to differentiate the candidate cloud platforms. While there are potentially other more sophisticated test scenarios, the three test scenarios provided by CARE cover most of the usage scenarios of typical cloud applications. Hence, the CARE framework provides a set of test scenarios that strikes a good balance between simplicity and coverage.

- *End-user - cloud host* represents the scenario that an end-user accesses a web service application hosted on the cloud platform from a client side application. The response time would be the end-user's primary concern in terms of the cloud application performance.

- *Cloud host - cloud database* represents the scenario that an end-user operates on a form or an article hosted in the cloud database through the cloud hosting server. The time taken to send the request from the end-user to the cloud host server is excluded as the focus is on the impact of different data sizes on the database processing time. It is especially interesting to be able to measure the database processing time of concurrent request that have been simultaneously generated by thousands of end-users. The database contention due to concurrent requests will be a key-determining factor in the overall scalability of the cloud platform in this type of scenario. Besides identifying different performance characteristics across cloud databases, a local database (LocalDB) is also provided by the CARE framework in a cloud hosting server as a reference point for comparison to other cloud databases.

- *End-user - cloud database* illustrates a large file transfer scenario. It is conceivable that data-intensive computing would be increasingly pervasive in the cloud where a large variety of new media content, such as video, music, medical images, and etc, would be stored and retrieved from the cloud. Understanding the characteristics of cloud and associated network behavior in handling *big data* is an important contribution towards improving the ability to better utilize cloud computing to handle such data.

3.1.3 Load test strategies

The CARE framework supports two types of load test strategies: high stress test strategy and low stress test strategy. The different load test strategies are applied across the various test scenarios listed in Section 3.1.2, in order to provide a more comprehensive evaluation and comparison.

The low stress test strategy sends multiple requests from the end-user side in a sequential manner. This is appropriate for simulating systems where there is a single or small number of end-users. It also provides a reference point for comparison to the high stress test strategy and also for obtaining base network latency benchmarks.

The high stress test strategy provides simulated concurrent requests to cloud platforms in order to obtain key insights on the cloud architecture, particularly for observing performance behavior under load.

Figure 3.2 illustrates the workflow of the high stress test strategy. The configurable parameter called *repeating rounds* is set to 6 by default. This represents the warm-up period, where there is typically a large performance variation due to certain phenomena such as cloud connection time. The performance results arising from the warm-up time stage are discarded by the performance results compilation framework, in order to produce more repeatable and stable testing results. Another configurable parameter *concurrent threads* is set to start at 100 by default. It is then incremented by another configurable parameter *increment* after every round of testing, the CARE framework currently sets the default value to 200 for the high stress test strategy, and 0 for the low

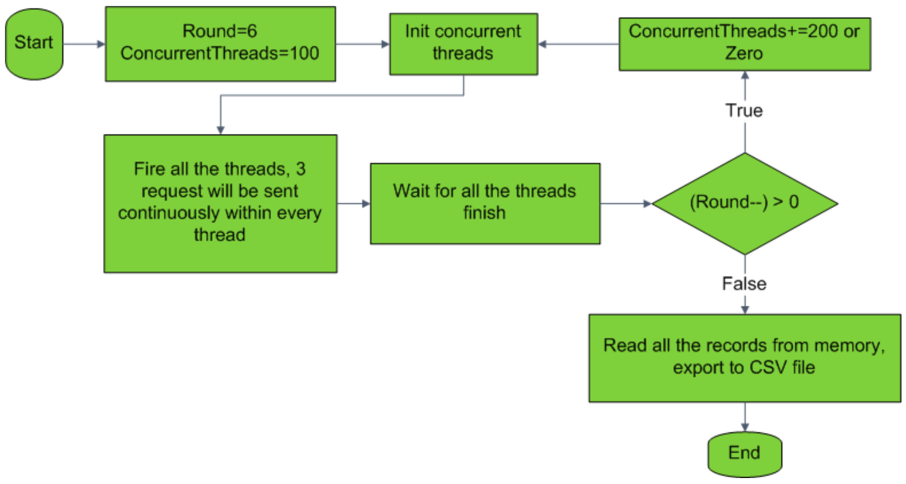

Start
Round=6 ConcurrentThreads=100
Init concurrent threads
ConcurrentThreads+=200 or Zero
True
Fire all the threads, 3 request will be sent continuously within every thread
Wait for all the threads finish
(Round--) > 0
False
Read all the records from memory, export to CSV file
End

Figure 3.2: The flow chart of evaluation strategies

stress test strategy. For example, for the high stress test strategy, after the initial 6 rounds, the number of concurrent threads fired by one end-user would go from 100 to 300, 500, 700, 900 and 1100 in successive rounds. Therefore, a maximum of 3300 concurrent threads can be achieved since 3 end-users are applied in the evaluation.

For the high stress test strategy, a number of continuous requests are sent within every thread to maintain its stress on the cloud platform over a period of time. If only a single request is sent to the cloud in each thread, our observation is that the expected concurrent stress cannot always be reached, and due to network latency and variability, the arrival time and order of packets at the cloud platform can vary widely. Hence in the CARE framework, another configurable parameter *continuous request* is provided with a default value of 3, striking a balance of providing a more sustained and even workload to the cloud and enabling the test to be conducted across different concurrent clients.

Lastly, as cloud computing is essentially a large-scale shared system, where the typical cloud end-user would be using a publicly shared network in order to access cloud services, it must be that there can be variations in network capacity, bandwidth, and latency issues, that fluctuates over time. The CARE framework thus provides a scheduler that support scheduled *cron*[1] jobs to be automatically and repeatedly activated to retrieved testing samples across different times over a 24 hour period.

The flow chart of the low stress test strategy for requests is essentially a simplified version of the high stress strategy shown in Figure 3.2, with the difference being that the multi-threaded functions are deactivated.

3.1.4 Building a test set with CARE

By using the CARE framework, it is possible to combine the various test scenarios with the various load test strategies to produce a comprehensive test set.

While the test set can be designed and created using the CARE framework depending on the precise test requirement, the CARE framework also comes with a reusable test set that aims to provide the test coverage of a large number of commonly found cloud application types. Table 3.1 illustrates a view of all test sets.

Firstly, there are five Contract-First Web Service based test methods, namely high stress round-trip, low stress database read and write, and high stress database read and write. There are also three RESTful Web Service based methods, low stress large file

Table 3.1: Building a test set

Test Set Method	Test Scenario	Load Test
High stress round-trip	End-user - cloud host	High stress test strategy
Low stress database read and write	Cloud host - cloud database	Low stress test strategy
High stress database read and write	Cloud host - cloud database	High stress test strategy
Low stress large file read, write, and delete	End-user - cloud database	Low stress test strategy

[1]`http://linux.die.net/man/8/cron`

read, write and delete, respectively. The four key methods in the test set are listed in Table 3.1.

- *High stress round-trip*: The end-users concurrently send message requests to cloud hosting servers. For each request received, the servers immediately echo back to the end-users with the received messages. The response time is recorded in this test. This is the base test that provides a good benchmark for a total round trip cloud application usage experience as the response time as experienced by the average end-user will be affected by the various variable network conditions. This is a useful test to indicate the likely end-user experience in an end-to-end system testing scenario.

- *Low stress database read and write* uses the cloud host - cloud database scenario. It starts with the low stress test strategy, which provides an initial reference result set for subsequent high stress load tests. This test is performed with varying data sizes, representing different cloud application data types. The data types provided by the CARE framework are: a single character of 1 byte, a message of 100 bytes, an article of 1 kilobyte, and a small file of 1 megabyte. These data types are sent along with the read or write requests, one after another to the cloud databases via the cloud hosting servers. The database processing time will be recorded and then returned to the end-user within the response. In terms of request size the CARE framework follows the conventional cloud application design principle of storing data that are no larger than 1 kilobyte in structured data oriented storage, namely Amazon SimpleDB and Microsoft Windows Azure Table Storage. Data that are larger than 1 kilobyte will be put into binary data oriented databases, including Amazon S3 and Microsoft Windows Azure Blob Storage. In addition, Google App Engine Datastore supports both structured data and binary data in the same cloud database.

- *High stress database read and write* are based on the high stress test strategy. It simulates multiple read/write actions concurrently. The number of concurrent requests range is configurable, as described in Section 3.1.3. Due to some common cloud platform quota limitations, for example Google App Engine by default limits incoming bandwidth to a maximum of 56 megabytes per minute, this test uses a default test data size of 1 kilobyte. This test data size can be configured to use alternative test data sizes if the target testing cloud platform does not have those quota limitations. Lastly, a cron job is scheduled to perform the stress database test repeatedly over different time periods across the 24 hour period.

- *Low stress large file read, write, and delete* are tests designed to evaluate large data transfer in the end-user - cloud database scenario. The throughput measure is as observed by the end-user. Once again, this test aims to characterize the total end-to-end large data handling capability by the cloud platform, taking into consideration the various network variations. The CARE framework provides some default test data: ranging from 1 megabyte, 5 megabytes, 10 megabytes, and through to 15 megabytes. A RESTful Web Service based end-user is implemented for a set of target cloud databases, including Amazon S3 and Microsoft Windows Azure Blob Storage. Note that the CARE framework does not provide a test for the Google App Engine, as Google App Engine Datastore does not support an interface for direct external connection for large file access.

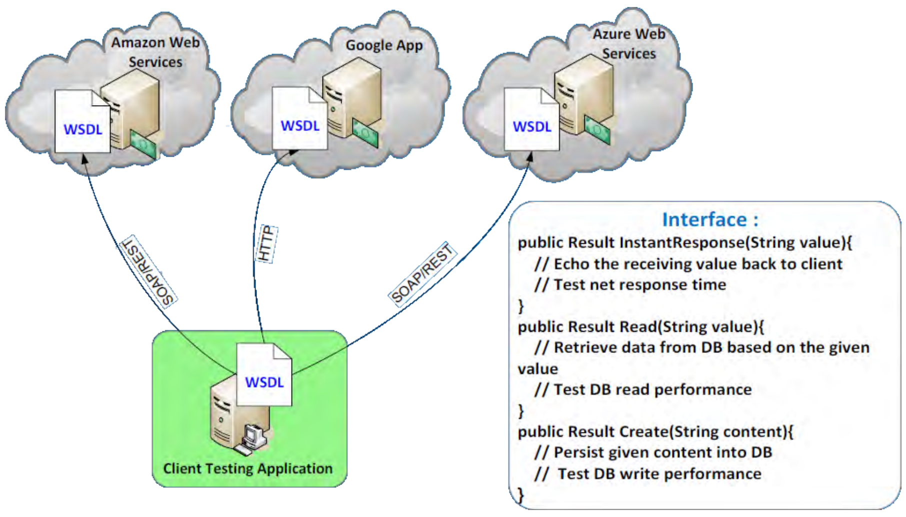
3.2 Application of CARE to cloud platform evaluation

Providing a common reusable test framework across a number of different clouds is a very challenging research problem. This is primarily due to the large variations in architecture, service delivery mode, and functionality provided across various cloud platforms, including Amazon Web Services, Google App Engine, and Microsoft Windows Azure. Firstly, the service models of cloud hosting servers are different: Amazon EC2 uses the infrastructure as a service model; Google App Engine uses the platform as a service model; while Microsoft Windows Azure combines both the infrastructure as a service and platform as a service models. Different service models have different levels of system privileges and different system architectures. Moreover, the connections among cloud hosting servers, cloud databases and client applications tend to utilize different protocols, frameworks, design patterns and programming languages which all add to the complexities to the task of providing a common reusable evaluation method and framework.

Therefore, we proposed a unified and reusable evaluation interface based on Contract-First Web Services and RESTful Web Services, for the purpose of keeping as much commonality as possible. As illustrated in Figure 3.3, for the Contract-First Web Services: a WSDL file is firstly built; then, the cloud hosting servers implement the functions defined in this WSDL file; lastly, a unified client interface is created from the WSDL file which allows communication via the same protocol, despite of existing variants. While for RESTful Web Services, direct access to cloud databases is made without passing the cloud hosting servers. The CARE framework currently provides the reusable common client components, and the cloud server components for Microsoft Windows Azure, Google App Engine and Amazon EC2.

The evaluation interface maximizes reusability of client application on the end-user side. The Contract-First Web Service based client application is able to talk to different cloud hosting servers via the same WSDL whereas a RESTful Web Service based client application can talk to cloud databases directly without passing the cloud hosting servers via the standard HTTP protocol.

The evaluation interface hides variations on the cloud side. As discussed in Sec-



```
public Result InstantResponse(String value){
   // Echo the receiving value back to client
   // Test net response time
}
public Result Read(String value){
   // Retrieve data from DB based on the given
value
   // Test DB read performance
}
public Result Create(String content){
   // Persist given content into DB
   //  Test DB write performance
}
```

Figure 3.3: Contract-First Web Service based client application

tion 2.2, the underline design of the three cloud platforms are different from each other. The Contract-First Web Services hide heterogeneous implementation of each cloud platform: Tomcat 6.0, Apache CXF, and a local PostgreSQL database are used on a small Ubuntu-based instance in Amazon EC2; Windows Communication Foundation (WCF) and C# codes are used on Microsoft Windows Azure; while Python-based ZSI and Zope Interface frameworks are used in Google App Engine. However, it is noted that potential performance difference is inevitable due to different programming languages. Thus, the CARE framework cloud server components follow the design principle of always using the native/primary supported language of the cloud platform in order to build the most optimal and efficient test components for each cloud platform.

3.3 Experiment results and exception analysis

In this section, quantitative results of four test set methods will be examined. Moreover, exceptions and errors captured during the evaluation will be analyzed by considering the results as an average over all test results. Some environmental information for the conducted tests are noted here:

- The client environment executing the CARE evaluation strategy runs on 3 Debian machines with Linux kernel 2.6.21.6-ati. Each evaluation machine is a standard Dell Optiplex GX620, equipped with Intel Pentium D CPU 3.00 GHz, 2 GB memory, and 10/100/1000 Base-T Ethernet.
- Both Amazon EC2 and Microsoft Windows Azure instances use the default type, small instance with single core.
- The sample test results listed here were conducted during the period of April - June 2009.

3.3.1 Qualitative experience of development utilities

In Amazon EC2, an administration role will be granted to developers when a virtual machine instance is created. This allows the developers to install whatever they want in the instance. In other words, there is no restriction on selecting development environments for Amazon EC2. But on the other hand, being able to select different work needs to be done, such as uploading and installing the required runtime environments for the application.

The key highlights of the Microsoft Windows Azure platform are its heavily equipped frameworks and environments. Almost all existing Microsoft web development frameworks and runtime environments are supported in Microsoft Windows Azure. As a result of this, developers can simply focus on the business logic implementation with C# or PHP. But the key downside is that they have to stick with Microsoft development environments, Microsoft Visual Studio.

In contrast to Microsoft Windows Azure which offers fully functioned frameworks, and Amazon EC2 which provides highly configurable environment, Google App Engine re-implements programming languages to suit the different development approaches.

Google has currently enabled Python and JVM-supported languages on its cloud platform where developers are free to choose frameworks based on Python and JVM-supported languages to improve their productivity. But, in practice, there are some limitations on the Google App Engine which restrict the range of choices, such as no multiple threads, no local I/O access, and 30 seconds timeout a request handler. Additionally, Google also offers other Google APIs to integrate Google App Engine with other Google services.

3.3.2 Quantitative results of test sets

High stress round-trip

Figures 3.4 to 3.6 indicate the cumulative distribution function of response time under varying amount of concurrent stress requests, which range from 300, 900, 1500, 2100, 2700, up to 3300 requests respectively.

The observation of three cumulative distribution functions confirms that the larger the requests, the longer the response time will be. But the incremental step of response time varies from one group of requests to another, depending on the cloud hosting servers. For 80% of cumulative distribution functions, the response time of Amazon EC2 in Figure 3.4 and Microsoft Windows Azure in Figure 3.5 are dramatically increased at 1500 requests and 900 requests respectively. For Google App Engine in Figure 3.6, although the response time shows an increasing trend, there is no significant leap between neighboring groups of requests.

The reason for these observations could be explained from the scalability aspect. If response time increases steadily and linearly under stress in Google App Engine, there is certainly some good scalability capability as its cloud hosting server is thread based, allowing more threads to be created for additional requests. Nevertheless, the cloud hosting servers of Amazon EC2 and Microsoft Windows Azure are instance based. The computing resources for one instance are preconfigured and more resources for additional requests cannot be obtained unless extra instances are deployed.

Low stress database read and write

In Figure 3.7, the average database processing time of reading 1 byte, 100 bytes, and 1 kilobyte are within 50 milliseconds, while the database processing time of writing small size data in Figure 3.8 varies from 10 milliseconds to 120 milliseconds. From this, it is obvious that for each cloud database, the reading performance is faster than the writing performance for the same amount of data. The two figures also state that the local database in Amazon EC2 instance shows its strength for message sizes that ranges from 1 byte to 1 kilobyte. As the evaluation environment is low stress, and as such, the cloud host is not under load, so it is consistent that the local database without any optimizations can handle requests effectively. The latency from the cloud hosting server to the local database is also smaller, since they are in the same Amazon EC2 instance.

When the size of request reaches 1 megabyte, Amazon S3, shown as orange dots in figures, almost has the same write performance as Google App Engine Datastore, but the former is almost three times slower than the latter in reading. Microsoft Windows Azure Blob Storage, shown as green triangles in figures, takes less time than the others in both reading and writing.

The cumulative distribution functions of read and write throughput in cloud databases demonstrated similar behavior as in Figures 3.9 and 3.10. Moreover, for the 1 megabyte database reading and writing test, the cumulative distribution functions also show that approximately 80% of requests are processed at 10 megabytes per second.

High stress database read and write

In this test, the number of concurrent requests in the evaluation varies from 300 to 3300 with step increments of 300. The collection of database processing time of each cloud

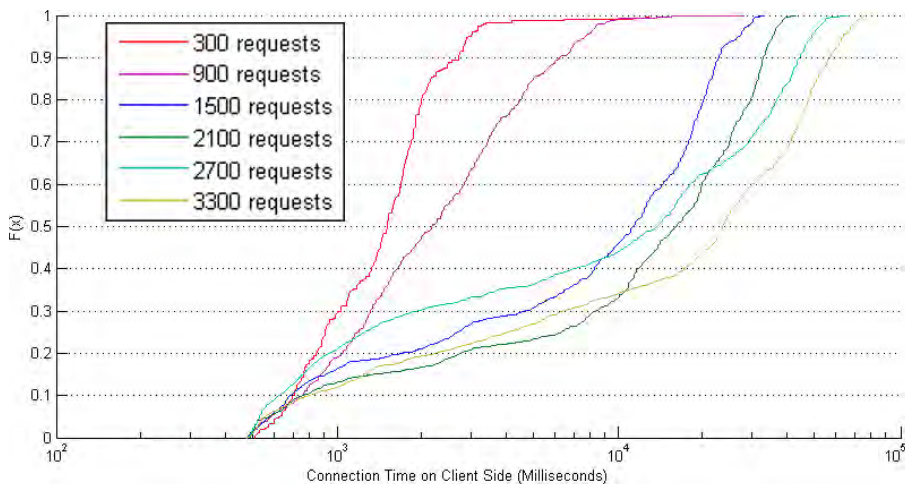


Figure 3.4: The cumulative distribution function of high stress round-trip between the end-user and the Amazon EC2 cloud hosting servers

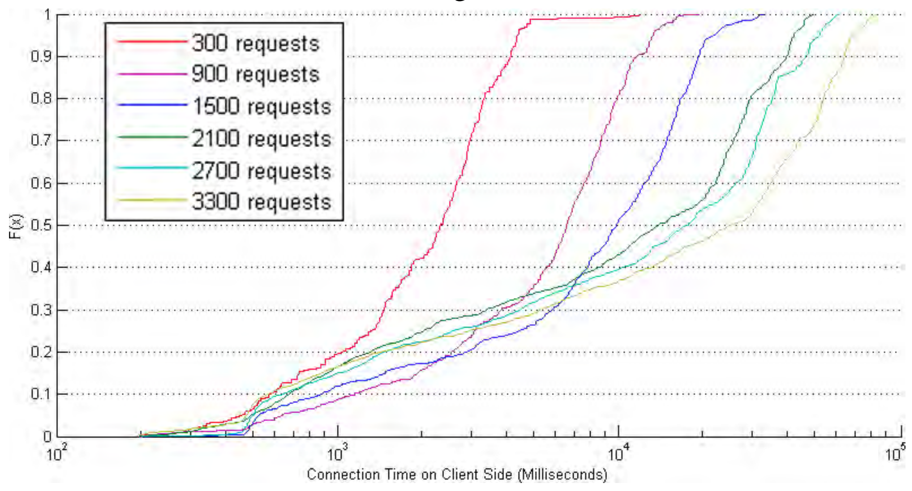


Figure 3.5: The cumulative distribution function of high stress round-trip between the end-user and the Microsoft Windows Azure cloud hosting servers

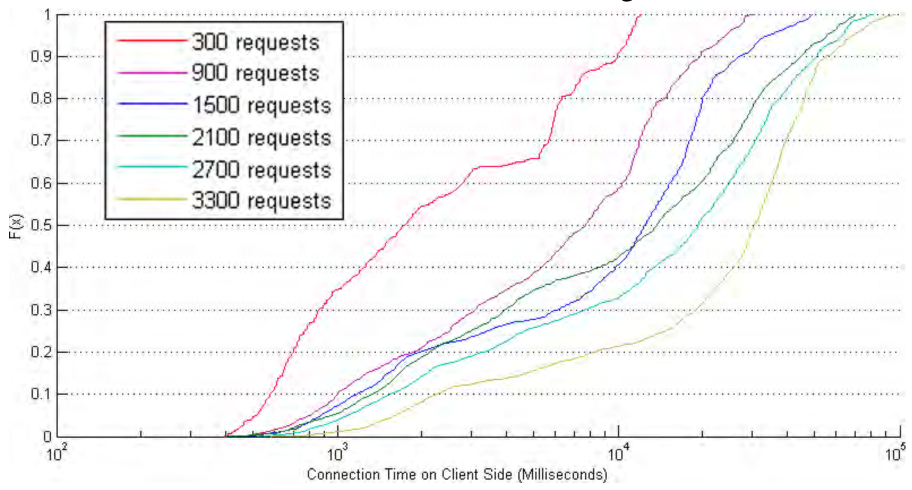


Figure 3.6: The cumulative distribution function of high stress round-trip between the end-user and the Google App Engine cloud hosting servers

database under 2100 concurrent requests are shown in Figure 3.11. From 2100 concurrent requests onwards, cloud host servers started to produce errors, these are listed in detail in Tables 3.3 and 3.4 in Section 3.3.3.

Instead of being the best performer as in low stress database read and write, the local database in Amazon EC2 now performs the worst among all platforms. It implies the poor capability of handling concurrent requests within the same instance as the compute capability. Moreover, Google App Engine Datastore, Amazon SimpleDB and Microsoft Windows Azure Storage all continue to show faster speeds in read operations than write operations.

Low stress large file read, write, and delete

Figure 3.12 shows the average database processing time of reading, writing and deleting binary files in the cloud databases directly. It can be seen that reading, shown in the left figure, is faster than writing, shown in the middle figure, in general. Both database processing time of read and write for Amazon S3 and Microsoft Windows Azure Blob Storage are linearly increasing with increasing proportion of data size. It is likely the limitation of the local network environment will come before getting insights of the

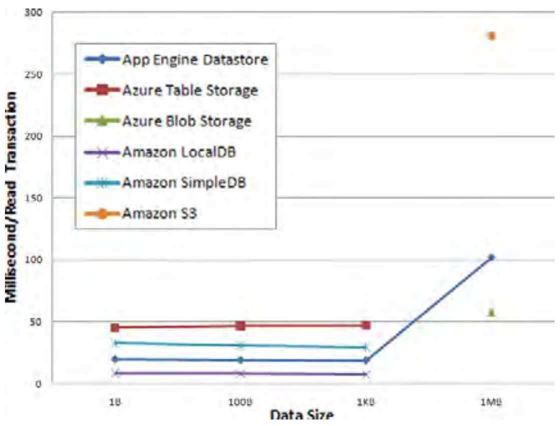


Figure 3.7: The average read time in cloud databases with low stress database read test set

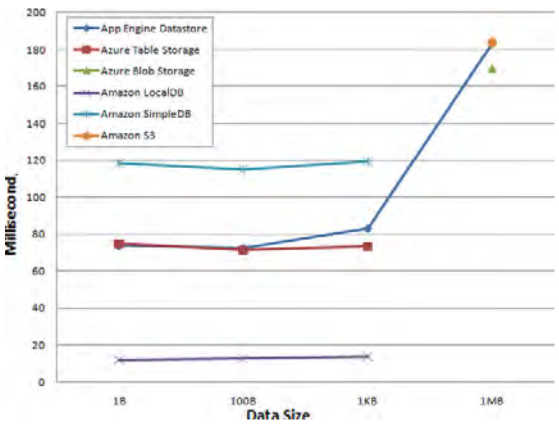


Figure 3.8: The average write time in cloud databases with low stress database write test set

cloud databases. This is why the CARE framework provides a range of scenarios, for example, end-user - cloud database, as well as cloud host - cloud database, so that the performance characteristics can be evaluated with and without the network variations and effects in place.

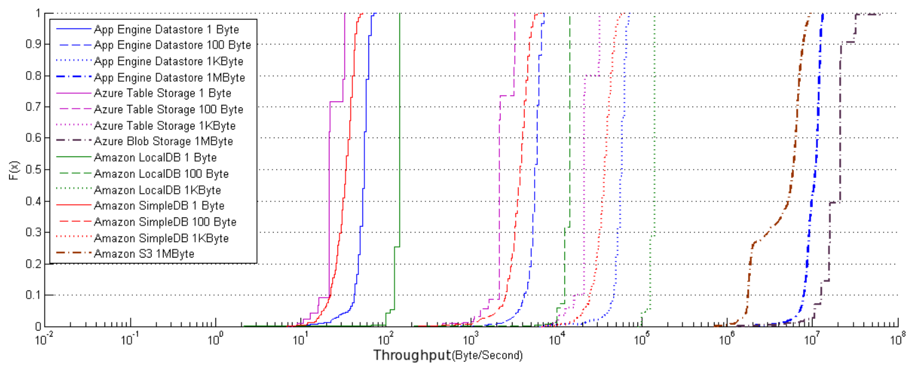


Figure 3.9: The cumulative distribution function of read throughput in cloud databases with low stress database read test set

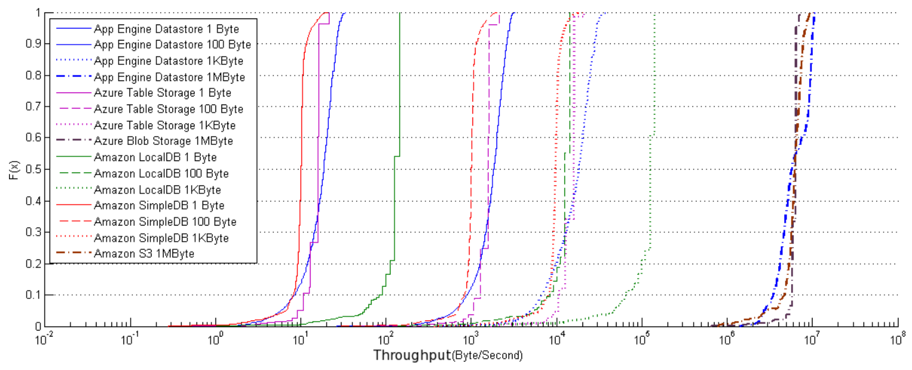


Figure 3.10: The cumulative distribution function of write throughput in cloud databases with low stress database write test set

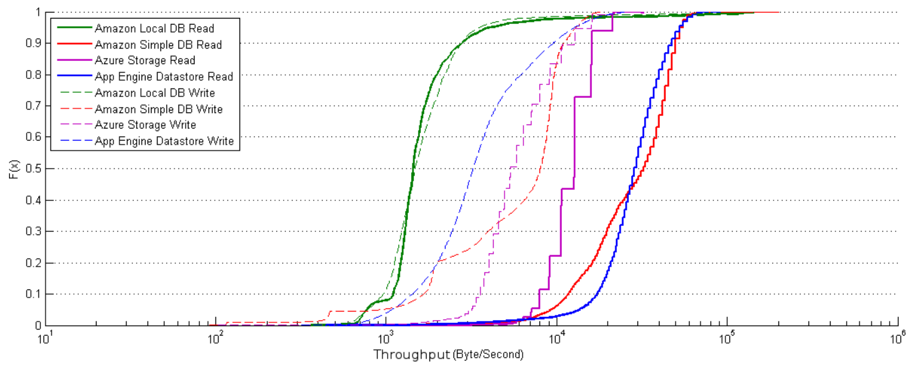


Figure 3.11: The cumulative distribution function of read and write throughput in cloud databases with high stress database read and write test sets

The average database processing time of the delete operation, shown in the right figure, is interesting as the observation shows a constant result regardless of data sizes. It is confirmed that neither Amazon S3 nor Microsoft Windows Azure Blob Storage will delete data entries on the fly. Both of them mark the entity and reply with successful request message at the first instant where the actual delete operation will be completed afterwards.

3.3.3 Exception analysis and error details

Overall error details

All error messages and exceptions were logged and captured by the CARE framework. This is a useful feature for carrying out offline analysis. The observations show that all errors occurred during the high stress database read and write tests. The CARE framework also logs the errors/exceptions according to various categories:

- *Database error* happens during the period of processing in cloud databases.
- *Server error* occurs within cloud hosting servers, for instance, not being able to allocate resources.
- *Connection error* is encountered if a request does not reach cloud hosting servers due to network connection problems, such as package loss and proxy being unavailable.

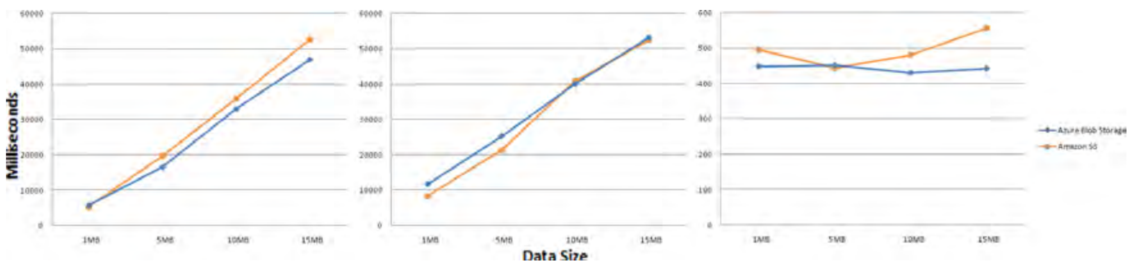


Figure 3.12: The database processing time of read, write, and delete in cloud databases with low stress large file read, write, and delete test sets

Table 3.2: Total error detail analysis

Category	Error Messages	Reasons	Locations
Database error	datastore_errors: Timeout	Multiple action perform at the same entry, one will be processed others will fail due to contention	Google Datastore
		Request takes too much time to process	Google Datastore
	datastore_errors: TransactionFailed-Error	An error occurred for the API request datas-tore_v3.RunQuery()	Google Datastore
	apiproxy_errors: Error	Too much contention on datastore entities	Google Datastore
	Amazon SimpleDB is currently unavailable	Too many concurrent requests	Amazon SimpleDB
Server error	Unable to read data from the transport connection	WCF failed to open connection	Microsoft Windows Azure
	500 Server Error	HTTP 500 ERROR : Internal Error	Google App Engine
	Zero Sized Reply		Amazon EC2
Connection error	Read timed out	HTTP time out	Microsoft Windows Azure/ Amazon EC2
	Access Denied	HTTP 401 ERROR	Microsoft Windows Azure/ Google App Engine/ Amazon EC2
	Unknown Host Exception		Microsoft Windows Azure
	Network Error (tcp_error)	Local proxy connection error	Microsoft Windows Azure/ Google App Engine

In general, a response with connection error is classified as an incomplete request; and a request to server error or database error is classified as a failed request. The error details of each category are listed in Table 3.2.

Average errors over different time periods

The CARE framework is also able to produce unavailability information based on error and exceptions logs over a long period of time. Table 3.3 and Table 3.4 show different average error rates of high stress database read and write methods over different time periods. As shown in the table, both read and write connection error rates of the local database in Amazon EC2 and Google App Engine Database vary in a range from 15% to 20%. This figure is highly variable over the 24-hour period especially as it is subjected to network conditions, as well as the health status of the cloud server. Amazon SimpleDB achieves the lowest error rates for both reading and writing operations with an error average of less than 10%, with average reading error rate that approaches 0%. On the contrary, Microsoft Windows Azure Table Storage has the highest reading error rate of more than 30%.

In spite of read and write connection error rates, average successful read request rates are high at almost 99.99% of completed request. Although Google Datastore and Amazon SimpleDB responded with write database error for 31.67 and 111.17 times respectively, the successful write request rates are generally high, with the worst one logging at more than 99.67% of completed request.

Among all cloud hosting servers, Google App Engine exhibits the most number of server errors where most errors were 500 Server Error messages. The largest group of server errors happened after May 20 23:30:00 PST 2009. Meanwhile, some significant latency started appearing in the Google App Engine's overall system status dashboard around one or half an hour earlier than the given time. It is likely that the significant

Table 3.3: Average error (rates) of high stress database read over different time periods

Cloud databases	Database error	Server error	Connection error	Successful request
Amazon SimpleDB	0.00 (0.000%)	0.00 (0.000%)	41.00 (0.127%)	32,359.00 (99.873%)
Amazon LocalDB	0.00 (0.000%)	16.40 (0.051%)	6368.40 (19.656%)	26,015.20 (80.294%)
Microsoft Windows Azure Table Storage	0.00 (0.000%)	0.00 (0.000%)	11,593.80 (35.783%)	20,806.20 (64.217%)
Google Datastore	2.25 (0.007%)	4.75 (0.015%)	5462.75 (16.860%)	26,930.25 (83.118%)

latency of the overall Google App Engine system could be a cause of the server errors in the experiment. However, there is no direct evidence to prove such a causality.

Average connection error rates under different loads

In high stress database read and write tests, as expected, the trend of the average connection error rates raises as the number of concurrent requests increases. Google Datastore via Google App Engine and Amazon SimpleDB via Amazon EC2 have a smaller percentage trend in reading than writing, while Microsoft Windows Azure Table Storage and the local database in Amazon EC2 on the contrary, display higher rates in read operations than write operations.

Amazon SimpleDB via Amazon EC2 maintains the lowest error rates in both reading and writing, almost approaching 0% in read tests. While the local database via Amazon EC2, which shares the same instance with the web application of Amazon SimpleDB via Amazon EC2, started receiving a high percentage of connection errors from 1500 concurrent requests. The reason of this phenomenon could be explained by that the local database causes additional resource contention by virtually being inside the same instance as the host server instance. This leads to a less scalable architecture, as a trade-off to smaller latency from host server to cloud database.

For Microsoft Windows Azure, the connection error percentage begins to leap, from less than 1% at 1500 requests, to more than 50% and 30% in reading and writing separately at 3300 concurrent requests. This indicates that a limit in terms of what this Azure server instance can handle has been hit.

For Google App Engine, a large number of connection errors under high load has been observed. Most connection errors from Google App Engine contain the access denied message, which is a standard HTTP 401 error message. Through cross checking the server side, there is no record of HTTP 401 at all in the Google App Engine. This means that these requests are blocked before getting into the web application. The assumption can be made that the access is restricted due to a firewall in Google App Engine. When thousands of requests go into Google App Engine concurrently from the same IP, the firewall may be triggered. Upon some analysis of how App Engine manages incoming requests by using a HTTP traffic monitor, it is reasonable to conclude that this may be a security feature around to prevent denial of service attacks. There seems no way to get around of it, except reducing the number of requests.

3.4 Discussion

An empirical experiment was carried out to examine the effectiveness of CARE when applied to testing different cloud platforms. Results indicate CARE is a feasible approach by directly comparing three major cloud platforms, including cloud hosting servers and cloud databases. Analysis revealed the importance of acknowledging different service models, and that the scalability of cloud hosting servers is achieved in different ways. Horizontal scalability is available to some extent in Google App Engine, but is always restricted by the quota limitation. On the contrary, Amazon EC2 and Microsoft Win-

Table 3.4: Average error (rates) of high stress database write over different time periods

Cloud databases	Database error	Server error	Connection error	Successful request
Amazon SimpleDB	111.17 (0.343%)	9.50 (0.029%)	2470.83 (7.626%)	29,808.50 (92.002%)
Amazon LocalDB	0.00 (0.000%)	25.20 (0.075%)	5262.60 (16.243%)	27,112.20 (83.680%)
Microsoft Windows Azure Table Storage	0.00 (0.000%)	0.17 (0.001%)	4810.33 (14.847%)	27,589.50 (85.153%)
Google Datastore	31.67 (0.098%)	3037.37 (9.374%)	4787.50 (14.776%)	24,543.66 (75.752%)

dows Azure can only scale through manual work in which developers can specify rules and conditions for when instances should be added. This leads the classic trading off issue of complexity against scalability. Vertical scalability is not possible in Google App Engine since every process has to be finished within 30 seconds, and that it is beyond the control over the type of machines used for our application in the Google cloud. Where on the other hand, Amazon EC2 and Microsoft Windows Azure allow you to choose and deploy instances with varying sizes of memory and CPUs.

The unpredictable unavailability of cloud is of a greater issue, particularly for enterprise organizations with mission critical application requirements. Whilst bursts of unavailability are noticed, during the tests which are caused by a range of environmental factors, including variable network conditions. It is also observed that the cloud providers sometimes experience challenges in maintaining uninterrupted service availability. Despite sophisticated replication strategies, there is still a potential risk of data center breakdown even in the cloud, which may in turn affect the performance and availability of hosted applications. It is also noticed that at the time of writing, most cloud vendors provide an SLA availability of 99.9%, which is still some way away from the typical enterprise requirement of 99.999%.

The network condition makes a significant impact on the total performance and end-user experience for cloud computing. The performance of the end-to-end cloud experience highly relies on the network condition. If an end-user accesses cloud services through a poor network environment, it is not possible to take full advantage of the cloud platforms.

Chapter 4

Performance evaluation of database replication of NoSQL database as a service

NoSQL database as a service is part of the database as a service offering to complement traditional database systems often by removing the requirement ACID transactions as one common feature. NoSQL database as a service has been supported by many service providers that offer various consistency options, from eventual consistency to single-entity ACID. For the service provider, weaker consistency is related to a longer replication delay, and therefore should allow better availability and lower read latency.

This chapter investigates the replication delay of NoSQL databases by observing the consistency and performance characteristics of various offerings from the customers' perspective. The main contributions of this chapter are detailed measurements over several NoSQL databases, that show how frequently, and in what circumstances, different inconsistency situations are observed, and what impact the customers sees on performance characteristics from choosing to operate with weak consistency mechanisms. An additional contribution is the development of the overall methodology of experiments for measuring consistency from the customer's view. The chapter first presents an architecture for benchmarking various NoSQL databases in Section 4.1. Then, Section 4.2 reports on the experiments that investigate how often a read sees a stale value. For several platforms, data is always, or nearly always, up-to-date. For one platform, specifically Amazon SimpleDB, stale data is frequently observed. Thus, in Section 4.3, the performance and cost trade-offs of different consistency options are explored. Section 4.4 discusses some limitations of generalizing results and gives some conclusions.

4.1 Architecture of benchmark application

Figure 4.1 illustrates the architecture of the benchmark applications in this study. There are three roles composed: the NoSQL database, the *writer*, and the *reader*. A writer repeatedly writes 14 bytes of string data into a particular data element where the value written is the current time, so that it is easy to check which write is observed in a read. In most of the experiments that are reported, writing happens once every three seconds. A reader role repeatedly reads the contents from the data element and also notes the time at which the read occurs; in most experiments reading happens 50 times every second. Comparing read values reveals the probability of reading stale values over time. Assume a writer invokes a write operation at time t and a reader invokes a read operation at time $t + x$. "A period of time" to make replicas consistent is obtained by finding x when no stale value is observed.

In some experiments, the writer and reader roles are deployed as a single thread for the writer role, and single or multiple threads for the reader role, while in other experiments, a single thread takes both roles. For one experiment measurement, the writing and reading operations are run for 5 minutes, doing 100 writes and $15{,}000$ reads. The mea-

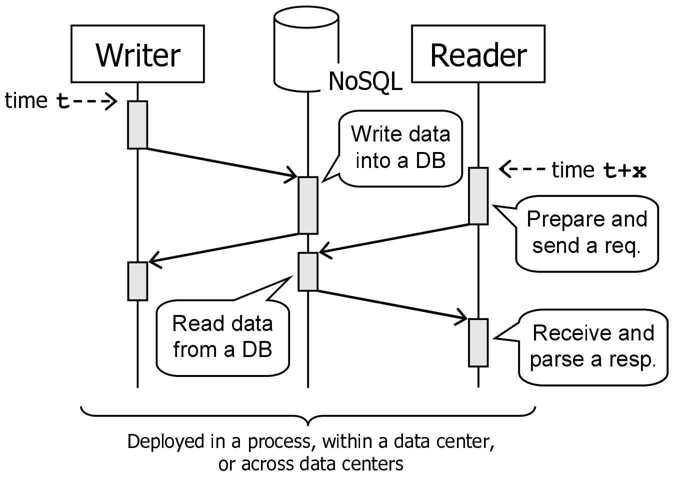


Figure 4.1: The architecture of NoSQL database as a service benchmark applications

surement is repeated once every hour, for at least one week, in October and November 2010. It must be noted that each measurement includes not only the processing time on NoSQL databases but also that of applications and network latency. In all measurement studies, it is confirmed that benchmark applications and networks are not performance bottlenecks.

In a post-processing data analysis phase, each read is determined to be either fresh or stale, depending on whether the value observed has a timestamp from the closest preceding write operation, based on the times of occurrence; also each read is placed in a bucket based on how much clock-time has elapsed since the most recent write operation. By examining all the reads within a bucket, from a single measurement run, or indeed aggregating over many runs, the probability of observing the freshest value by a read is calculated. Repeating the experiment through a week ensures that we will notice any daily or weekly variation in behavior.

4.2 Staleness of data on different cloud platforms

4.2.1 Amazon SimpleDB

Amazon SimpleDB is a distributed key-value store offered by Amazon. Each key has an associated collection of attributes, each with a value. For these experiments, a data element is taken to be a particular attribute kept for a particular key, which identifies, in SimpleDB terms, an *item*. SimpleDB supports a write operation call via *PutAttributes* and two types of read operations, distinguished by a parameter in the call to *GetAttributes*: *eventual consistent read* and *consistent read*. The consistent read is supposed to ensure that the value returned always comes from the most recently completed write operation, while an eventually consistent read does not give this guarantee. This study investigates how these differences appear to the customers who consume data.

Amazon SimpleDB is currently operated in several independent geographic regions and each of them offers a distinct URL as its access point. For example, `https://sdb.us-west-1.amazonaws.com` is the URL of SimpleDB operated in *us-west* region. It is used as the testbed in all experiments. The benchmark application for Amazon SimpleDB is implemented in Java and runs in Amazon EC2. It accesses SimpleDB through its REST interface. The writer writes timestamps, each of which is 14 bytes of string data, in a key-value pair. The reader reads a value from the same key-value pair using *eventual consistent read* or *consistent read* option. The study of Amazon SimpleDB comprises of both parts based on the access patterns. The access patterns determine the location options of EC2 instances that the writer and the reader could reside, including options of being in the same region or in different regions.

Access patterns

In the first pattern, the writer and reader run in the same single thread on an *m1.small* instance provided by Amazon EC2 with Ubuntu 9.10. The instance is deployed in the same region of SimpleDB, in the hope of minimizing the network latency. Although, it is not guaranteed that data items from SimpleDB will be in the same physical data center as the thread in EC2, using the same geographic region is the best mechanism to the customer to reduce network latency. For this access pattern, two consistency options, *read-your-write* and *monotonic read* are examined.

While in the second pattern, the writer and the reader are deliberately separated to multiple threads, with the following configurations:

1. A writer and a reader run in different threads but in the same process. In this case, read and write requests originate from the same IP address.

2. A writer and a reader run in different processes but in the same instance that is also in the same geographic domain as the data storage in *us-west* region. In this case, read and write requests still have the same IP address.

3. A writer and a reader run on different instances but both are still in the same region. In this case, requests originate from different IP addresses but from the same geographical region.

4. A writer and a reader run on different instances and different regions, one in *us-west* region and one in *eu-west* region. In this case, requests originate from different IP addresses in different regions.

The measurement is executed once every hour for 11 days from Oct 21, 2010. In total 26,500 writes and 3,975,000 reads were performed for accessing from a single thread. Since only one thread is used in the first study, the average throughput of reading and writing are 39.52 per second and 0.26 per second, respectively, where each measurement runs at least for five minutes. The same set of measurements was performed with eventual consistent read and with consistent read.

In the study of accessing from multiple threads and processes, each experiment was run for 11 days as well. In all four cases the probability of reading updated values shows a similar distribution as in Figure 4.2. Therefore, it is concluded that customers of Amazon SimpleDB see the same data consistency model regardless of where and how clients are placed. Hence, this section will focus on reporting observations of accessing from single thread with regards to two consistency options, read-your-write consistency and monotonic read consistency respectively.

Read-your-write consistency

Figure 4.2 shows the probability of reading the fresh value plotted against the time interval that elapsed from the time when the write begins, to the time when the read is submitted. Each data point in the graph is an aggregation over all the measurements for a particular bucket containing all time intervals that conform to millisecond granularity. With eventual consistent read the probability of reading the freshest data stays about 33% from 0 ms to 450 ms. It surges sharply between 450 ms and 500 ms, and finally reaches 98% at 507 ms. A spike and a valley in the first 10 ms are perhaps random fluctuations

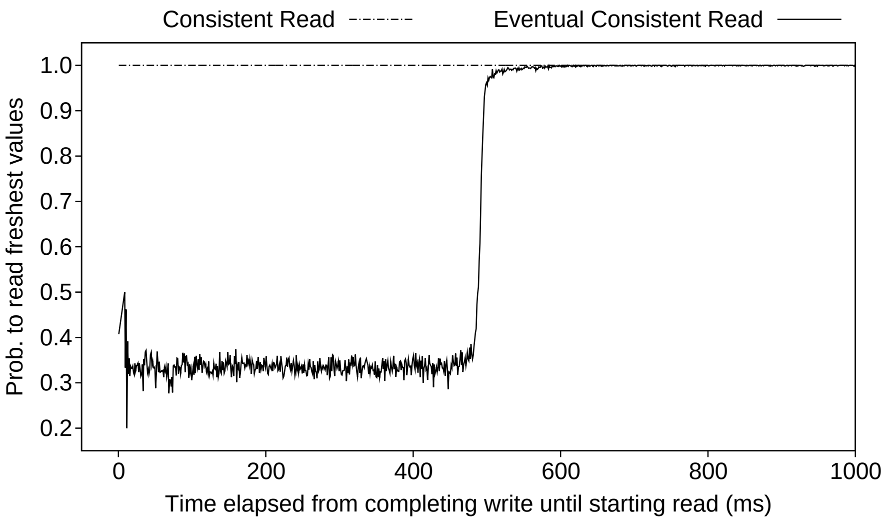


Figure 4.2: Probability of reading freshest value

due to a small number of data points. While with consistent read, the probability is 100% from about 0 ms onwards. To summarize further, Table 4.1 places all buckets whose time is in a broad interval together and shows actual numbers as well as percentages.

A type of relevant consistency is read-your-writes, which says that when the most recent write is from the same thread as the reader, then the value seen should be fresh. As stale eventual consistent reads are possible with Amazon SimpleDB within a single thread, so it is concluded that eventual consistent reads do not satisfy read-your-writes; however, consistent reads do achieve such level of consistency.

Moreover, the variability of the time is also examined when freshness is possible or highly likely, among different measurement runs. For eventual consistent reads, Figure 4.3 shows the first time when a bucket has the freshness probability of over 99%, and the last time when the probability is less than 100%. Each data point is obtained from a five minutes measurement run, so there are 258 data points in each time series. The median of the time to exceed 99% is 516.17 ms and coefficient of variance is 0.0258. There does not seem to be any regular daily or weekly variation, rather the outliers seem randomly placed. Out of the 258 measurement runs, 2nd and 21st runs show a non-zero probability of stale read after 4000 ms and 1000 ms respectively. Those outliers are considered to be generated by network jitter and other similar effects.

Table 4.1: Probability of reading freshest value

Time elapsed from starting write until starting read	Eventual consistent read	Consistent read
$[0, 450)$	33.40% (168,908/505,821)	100.00% (482,717/482,717)
$[500, 1000)$	99.78% (1192/541,062)	100.00% (509,426/509,426)

Monotonic read consistency

Monotonic read is an important consistency option (Vogels, 2009). It is defined as a condition where subsequent operations see data that is at least as fresh as what was seen before. This property can be examined across multiple data elements or for a single element as is considered here. The consistent read meets monotonic as it should be, since each read should always see the most recent value. However, eventual consistent read is not monotonic and indeed the freshness of a successive operation seems essentially independent of what was seen before. Thus, eventual consistent read also does not meet stronger consistency options such as causal consistency.

Table 4.2 shows the probability of observing fresh or stale values in each pair of successive eventual consistent reads performed during the range from 0 ms to 450 ms after the time of a write. The table also shows the actual number of observations out of $475{,}575$ of two subsequent reads performed in this measurement study. The monotonic read condition is violated, that is the first read returns a fresh value but the second read returns a stale value, in 23.36% of the pairs. This is reasonably close to what one would expect of independent operations, since the probability of seeing a fresh value in the first

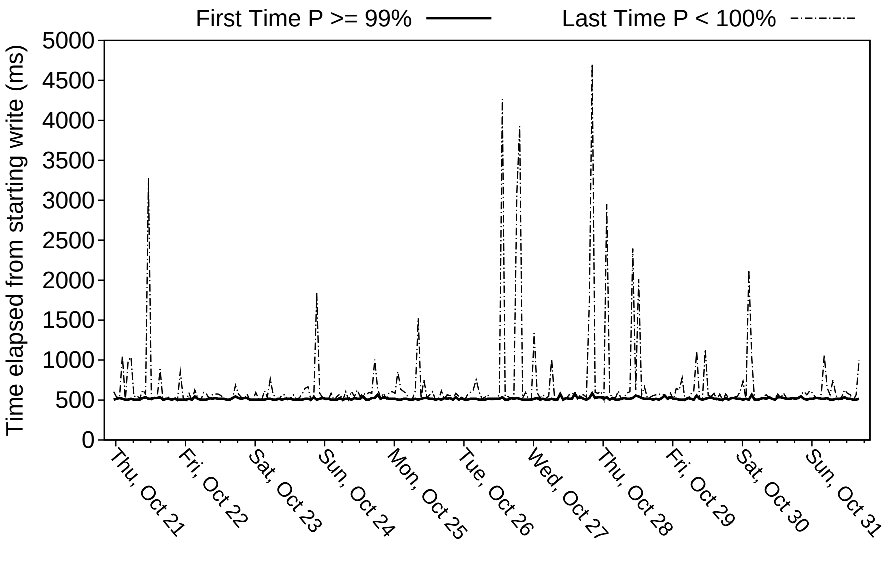


Figure 4.3: Time to see freshness with eventual consistent read

Table 4.2: Successive eventual consistent reads

First read \ Second read	Stale	Fresh
Stale	39.94% (189, 926)	21.08% (100, 1949)
Fresh	23.36% (111, 118)	15.63% (74, 337)

read is about 33% and the probability of seeing a stale value in the second read is about 67%. The Pearson correlation between the outcomes of two successive reads is 0.0281, which is very low, and it is concluded that eventual consistent reads are independent from each other.

4.2.2 Amazon S3

A similar measurement study was conducted on Amazon Simple Storage Service (S3) for 11 days. In S3, storage consists of objects within buckets, so our writer updates an object in a bucket with the current timestamp as its new value, and each reader reads the object. In this experiment, measurements for the same five configurations as SimpleDB's case are conduced, including a writer and a reader run in a single thread, different threads, different processes, different instances, and different regions. Amazon S3 supports two types of write operations, namely standard and reduced redundancy. A standard write operation stores an object so that its probability of durability is at least $99.999{,}999{,}999\%$, while a reduced redundancy write aims to provide at least 99.99% probability of durability. The same set of measurements was performed with both standard write and reduced redundancy write.

Documentation states that Amazon S3 buckets provide eventual consistency for overwrite *PUTS* operations. However, no stale data was ever observed in this study regardless of write redundancy options. It seems that staleness and inconsistency might be visible to a customer of Amazon S3 only in executions in the event of a failure in the particular nodes of the platform where the data is stored, during the time of their access; this is a very low probability event.

4.2.3 Microsoft Windows Azure Table Storage and Blob Storage

The experiment was also conducted on Microsoft Windows Azure Table Storage and Blob Storages for eight days. Since it is not possible to start more than one process on a single instance, specifically for a *web role* in this experiment, measurements for four configurations are conducted: a write and a reader run in a single thread, different threads, different instances or different regions. On Azure Table Storage a writer updates a property of a table and a reader reads the same property. On Azure Blob Storage a write updates a blob and a reader reads it.

The measurement study observed no stale data at all. It is known that all types of Microsoft Windows Azure Storages support strong data consistency (Krishnan, 2010) and this experiment confirms it.

4.2.4 Google App Engine Datastore

Similar to Amazon SimpleDB, Google App Engine Datastore keeps key-accessed entities with properties and it offers two options for reading: *strong consistent read* and eventual consistent read. However, the observed behavior for eventual consistent read in the Datastore is completely different from that of Amazon SimpleDB. It is known that the eventual consistent read of Datastore reads from a secondary replica only when a primary replica is unavailable. Therefore, it is expected that customers see consistent data in most reads, regardless of the consistency option they choose.

The benchmark application for Google App Engine Datastore is coded in Java and deployed in Google App Engine. Applications deployed in App Engine are not allowed to create threads; a thread automatically starts upon an HTTP request and it can run for no more than 30 seconds. Therefore, each measurement on App Engine runs for 27 seconds and measurements are executed every 10 minutes for 12 days. The same set of measurements was performed with strong consistent read and eventual consistent read. App Engine also offers no option to control the geographical location of applications. Therefore, only two configurations are examined: a writer and a reader are run in the same application, and a writer and a reader are run in different applications. Each measurement consists of 9.4 writes and 2787.9 reads on average, and in total $3{,}727{,}798$ reads and $12{,}791$ writes are recorded on average for each configuration.

With strong consistent read no stale value was observed. With eventual consistent read and both roles in the same application, no stale value was observed. However 11 out of $3{,}311{,}081$ readings, approximately $3.3 \times 10^{-4}\%$, observed stale values when a writer and an eventual consistent reader are run in different applications. It is hard to conclude for certain whether stale values might sometimes be observed when a writer and a reader are run in the same application. However, it suggests the possibility that Google App Engine offers read-your-writes level of eventual consistency. In any case, it is also clear that consistency errors are very rare.

4.3 Trade-off analysis of Amazon SimpleDB

In the hope of assisting the customer to make a well-informed decision about consistency options for reading data, the trade-off analysis could be made by considering consistency levels against response time and throughput, monetary cost, and implementation ideas, respectively. The benchmark architecture described in Section 4.1 is reused for the analysis. The measurement ran between 1 and 25 instances in *us-west* region to read and write one attribute, which is a 14 bytes string data, from an item in Amazon SimpleDB.

Each instance runs 100 threads, acting as emulated end-users, each of which executes one read or write request every second in a synchronous manner. Thus, if all requests' response time is below 1000 ms, the throughput of SimpleDB can be reported as 100% of the potential load. Three different read/write ratios were studied, including 99/1, 75/25, and 50/50 cases. Each measurement runs for five minutes with a set number of virtual machines, once every hour for one day.

4.3.1 Response time and throughput

As advised in Amazon SimpleDB FAQs[1], the benefits of eventual consistent read can be summarized as minimizing response time and maximizing throughput. To verify this advice, the difference in response time, throughput, and availability of the two consistency options is investigated, as the load is increased. Figure 4.4 shows the average, 95 percentile, and 99.9 percentile response time of eventual consistent reads and consistent reads at various levels of load. The result is obtained from the case of 99% read ratio and all failed requests are excluded. The result shows no visible difference in average response time. However, consistent read slightly outperforms eventual consistent read in 95 percentile and 99.9 percentile response time.

Figure 4.5 and Figure 4.6 show the average response time of reads and writes at various read/write ratios, plotted against the number of emulated end-users. A conclusion could be drawn that changing the level of replication intensity has a negligible impact on the read and write response times. Intuitively, it would be surprised that eventual consistent read does not outperform the consistent read as expected, but it is still reasonable if the possible implementation ideas, detailed in Section 4.3.3, are taken into consideration.

Figure 4.7 shows the absolute throughput, the average number of processed requests per second. Whiskers are plotted surrounding each average with the corresponding min-

[1]`http://aws.amazon.com/simpledb/faqs/`

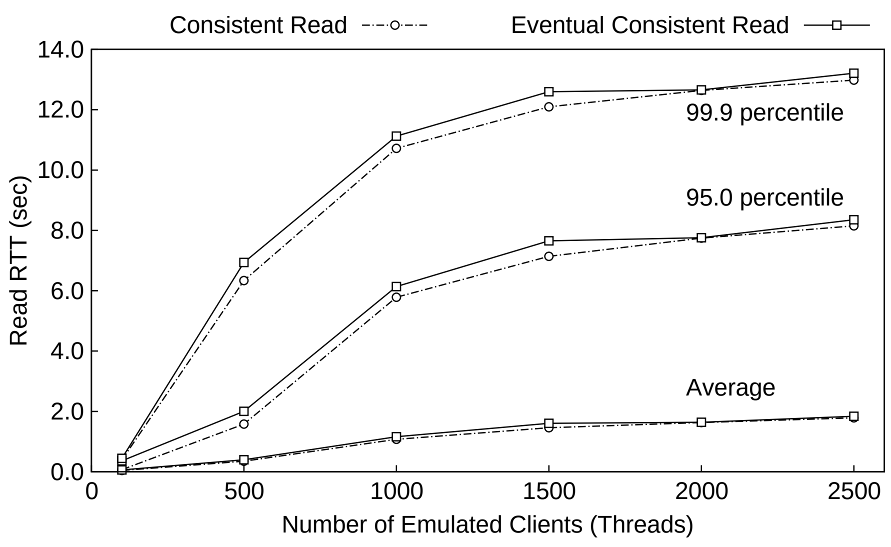


Figure 4.4: The average, 95 percentile, and 99.9 percentile response time of reads at various levels of load

imum and maximum throughput. Similar to the response time, consistent read results slightly outperforms that of eventual consistent read, though the difference is not significant. Figure 4.8 shows the throughput as a percentage of what is possible with this number of end-users. As the response time increased, each end-user sent less than one request every second and, therefore, the throughput percentage decreased.

It must be noted that Amazon SimpleDB often returns exceptions with status code 503, representing “Service is currently unavailable”, under heavy load. Figure 4.9 shows the average failure rates of eventual consistent reads and consistent reads, with each data point being marked with whiskers to highlight the corresponding maximum and minimum failure rates. Clearly the failure rate increased as the load increased, but again the observation is that eventual consistent read does less well than consistent read, although the difference is not significant.

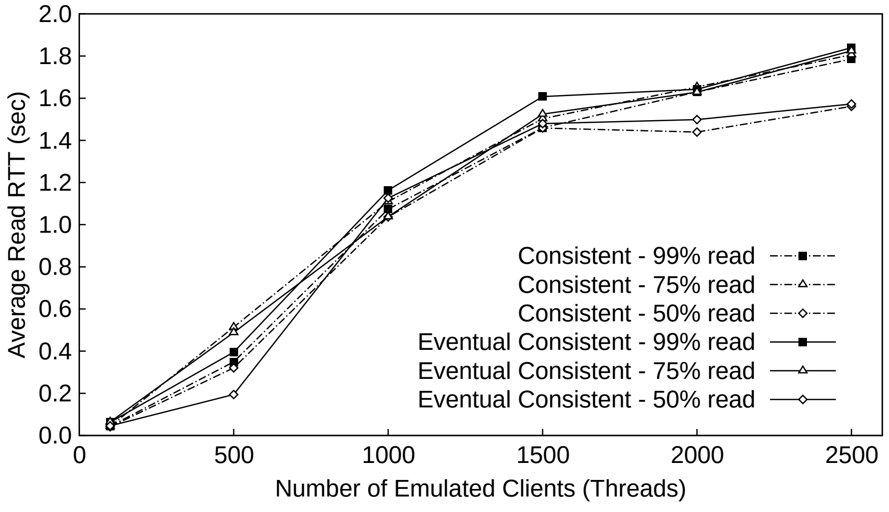


Figure 4.5: Response time of reads at various read/write ratios on Amazon SimpleDB

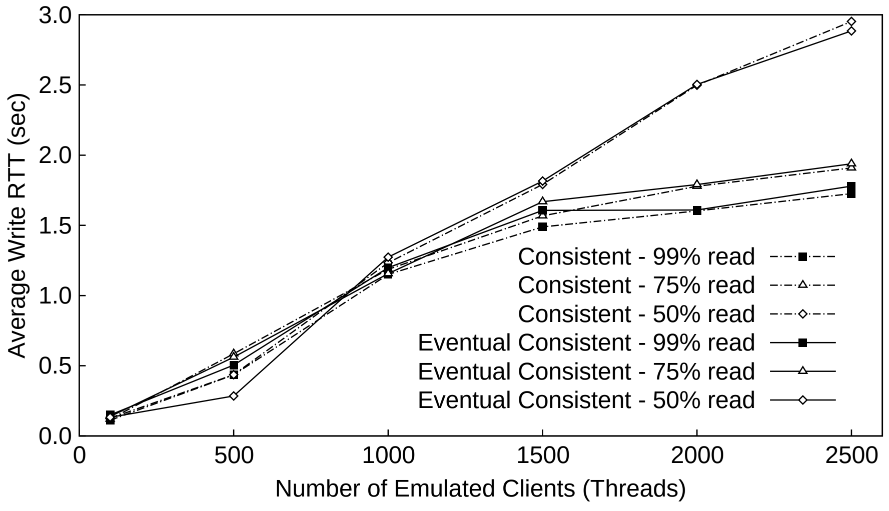


Figure 4.6: Response time of writes at various read/write ratios on Amazon SimpleDB

4.3.2 Monetary cost

A new perspective on which customers are usually concerned in the context of cloud computing is the trade-off against monetary cost. In *us-west* region, Amazon SimpleDB charges $\$0.154$ per SimpleDB machine hour, which is the amount of cost for using SimpleDB server capacity to complete requests, and therefore can vary depending on factors such as operation types and the amount of data to access. The monetary costs of two read consistency options for the runs described above are compared based on reported Sim-

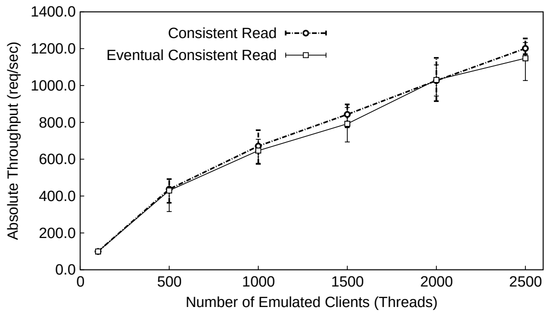


Figure 4.7: Processed requests of Amazon SimpleDB

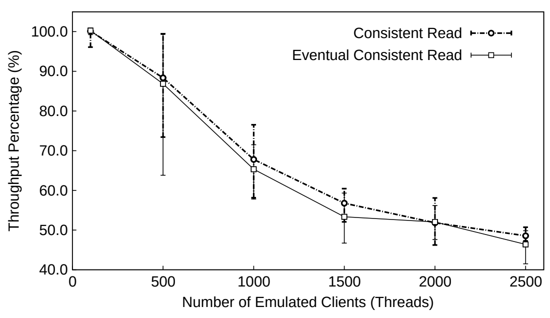


Figure 4.8: Throughput percentage of Amazon SimpleDB

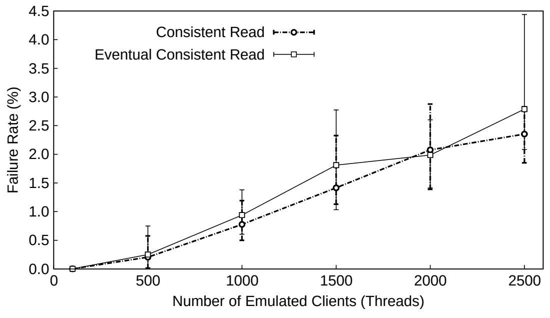


Figure 4.9: Request failure rate of Amazon SimpleDB

pleDB machine hour usage. Because the read operations of all runs constantly read 14 bytes string data from SimpleDB, the cost of read is constant, at $\$1.436$ per one million requests, regardless of the consistency options or workload. Also, the cost of write operations is constant at $\$3.387$ per one million requests as well, because the write operations of all runs always update SimpleDB with 14 bytes string data.

4.3.3 Implementation ideas

Although there is no published details about the implementation of Amazon SimpleDB, based on experiments, a few implementation ideas of SimpleDB can still be extracted. It seems feasible that Amazon SimpleDB maintains each item stored in 3 replicas, one primary and two secondaries. It is suspected that an eventually consistent read chooses one replica at random, and returns the value found there, while a consistent read will return the value from the primary. This aligns with previous experiment results showing the same latency and computational effort for the two kinds of read.

4.4 Discussion

This chapter reports on the performance and consistency of various cloud-based NoSQL storage platforms, as observed during some experiments. However, it is hard to say whether results can be extrapolated to predict expected experience for customers when using one of the platforms as all the usual caveats of benchmarks measurements still apply. For example, the workload may not be representative of the customers' needs, the size of the writes in the experiments is too small, and the number of data elements is small. Similarly, the metrics quoted may not be what matters to the customer as well, for example, the customer may be more or less skilled in operating the system; the experiments were not run for sufficiently long periods and the figures might reflect chance occurrences rather than system fundamentals.

Additionally, there are other particular issues when measuring cloud computing platforms. The cloud service provider moves on quickly and might change any aspect of hardware or software without providing sufficient advance notice to the customers. For example, even if the algorithm used by a platform currently provides read-your-writes, the cloud service provider could shift to a different implementation that does not provide the current guarantee. As another example, a cloud service provider that currently places all replicas within a single data center might implement geographical distribution, with replicas stored across data centers for better reliability. Such a change could happen without awareness of the customers, but it might lead to a situation where eventual consistent reads have observably better performance than consistent reads. Similarly, the background load on the cloud computing platforms might have a large impact, on latency or availability or consistency, but the customer cannot control or even measure what that load is at any time (Schad et al., 2010). For all these reasons, our current observations that eventual consistent reads are no better for the customer, might not hold true in the future.

Also taking the observations reported in this chapter as an example, The reported results are mainly obtained during October and November in 2011. Before that a similar experiments were conducted in May 2011 as well. By doing the comparison, most aspects were similar between the two sets of experiments, in particular the 500 ms latency till Amazon SimpleDB reached 99% chance for a fresh response to a read, the high chance of fresh data in eventual consistent reads in Amazon S3, Microsoft Windows Azure Blob Storage, and Google App Engine Datastore, and the lack of performance difference between SimpleDB for reads with different consistency. Other aspects had changed, for example in the earlier measurements there was less variation in the response time seen by reads on SimpleDB.

In order to achieve high availability and low latency, many NoSQL storage platforms drop the guarantee of strong consistency, by avoiding two-phase commit or synchronous access to a quorum of sites. Therefore, it is commonly said that developers should work around this by designing applications that can work with eventual consistency or similar weaker models. This chapter also examined the experience of the customer of NoSQL storage, in regard to weak consistency and possible performance trade-offs to justify its use, specifically by focusing on Amazon SimpleDB. This information should help a developer who is seeking to understand the new NoSQL storage platforms, and who needs to make sensible choices about choosing the right storage platform.

This chapter found that platforms differed widely in how much weak consistency is seen by customers. On some platforms, the customer is not able to observe any inconsistency or staleness in the data, over several million reads through a week. It seems that inconsistency is presumably possible, but are very rare. It might only happen if there is a failure of the NoSQL storage platforms. Therefore, the risk of inconsistency seems less important when compared to other sources of data corruption, such as bad data entry, operator error, customers repeating input, fraud by insiders, and etc. Any system design needs to have recourse to manual processes to fix the mistakes and errors from these other sources, and the same processes should be able to cover rare inconsistency-induced difficulties. On these platforms, it might be an option for the developer to sensibly treat eventual consistent reads as if they are consistent, accepting the rare errors as being unavoidable and thus its impact needs to be carefully managed.

On Amazon SimpleDB, the customer who requests eventual consistent reads experiences frequent stale reads. Also, this choice does not provide other desirable options like read-your-writes and monotonic reads. Thus the developer who uses eventual consistent reads must take great care in application design, to code around the potential dangers. However, in regard to no incentive in reducing latency, observed availability, and monetary cost, there is, in fact, no compensating benefit for the developer from choosing eventual consistent reads instead of using consistent reads. There may be benefits to the service provider when eventual consistent reads are done, but at present these gains have not been passed on to the customer. Thus on this platform in its current implementation, there is no significant monetary and performance benefits for a developer to code with eventual consistent reads.

Chapter 5

Performance evaluation of database replication of virtualized database servers

In general, virtualization technology is increasingly being used to improve the manageability of software systems and lower their total cost of ownership. Resource virtualization technologies add a flexible and programmable layer of software between applications and the resources used by these applications. One among several approaches for deploying data-intensive applications in cloud platforms, called the *virtualized database servers* approach, takes advantage of virtualization technologies by taking an existing application designed for a conventional data center, and then porting it to run on virtual machines in the public cloud. Such migration process usually requires minimal changes in the architecture or the code of the deployed application. In this approach, database servers, like any other software components, are migrated to run in virtual machines. One of the main advantages of this approach is that the application can have full control in dynamically allocating and configuring the physical resources of the database tier as needed. Hence, software applications can fully utilize the elasticity feature of the cloud environment to achieve their defined and customized scalability or cost reduction goals. In addition, this approach enables the software applications to build their geographically distributed database clusters. Without the cloud, building such in-house cluster would require self-owned infrastructure which represent an option that can be only afforded by big enterprises.

A common feature to the different cloud offerings of the NoSQL database as a service and the relational database as a service is the creation and management of multiple replicas of the stored data while a replication architecture is running behind-the-scenes to enable automatic failover management and ensure high availability of the service. In the previous chapter, experimental investigation of customer-based observations of the consistency, data staleness and performance properties of various cloud NoSQL databases have been carried out. In this chapter, virtualized database servers are the main target for exploration. The aim is to set a first yard stone in evaluating the performance characteristics of virtualized database servers in cloud environment. In particular, this chapter focuses on addressing the following questions with regards to the *master-slave* database replication strategy on Amazon EC2:

- How well does the master-slave replication strategy scale with an increasing workload and an increasing number of virtualized database replica servers in cloud? In principle, we try to understand what factors act as limits on achievable scale.
- What is the average replication delay or window of data staleness that could exist with an increasing number of virtualized database replica servers and different configurations to the geographical locations of the slave databases?

The remainder of this chapter is structured as follows. In Section 5.1, a few design decisions that are related to the benchmark application are explained, including customizing Cloudstone implementing fine-grained time/date function in MySQL, and applying clock synchronization in cloud. Meanwhile, Section 5.2 details the implementation of the experimental framework and the experimental environment. While the results of our experiments are presented in Section 5.3. Finally, the conclusion of the experiments are discussed Section 5.4.

5.1 Design of benchmark application

The Figure 5.1 shows the overall architecture of relational database as a service benchmark application. In general, it is a three-layer implementation. The first layer is a customized Cloudstone benchmark[1] which controls the read/write ratio and the workload. The second layer includes a master database that receives write operations from the benchmark and is responsible for propagating *writesets* to slaves. The third layer

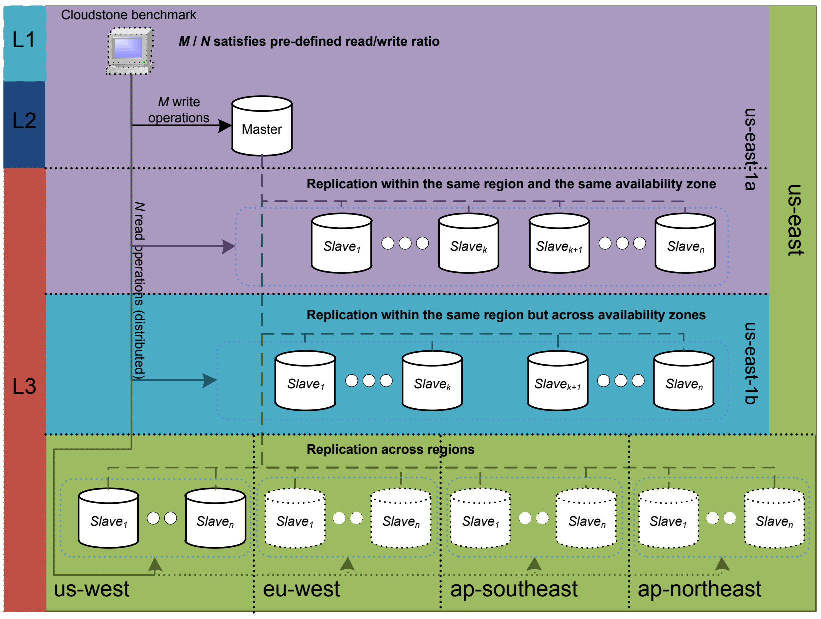

Cloudstone benchmark
L1
M / N satisfies pre-defined read/write ratio
L2
M write operations
Master
us-east-1a
us-east
Replication within the same region and the same availability zone
N read operations (distributed)
Slave1
Slavek
Slavek+1
Slaven
Replication within the same region but across availability zones
us-east-1b
L3
Replication across regions
us-west
eu-west
ap-southeast
ap-northeast

Figure 5.1: The architecture of relational database as a service benchmark application

[1]http://radlab.cs.berkeley.edu/wiki/Projects/Cloudstone

is a group of slaves which are responsible for processing read operations and updating writesets.

The design of the benchmark tool is relational-database-focused and replication-precision-driven. Therefore, there are several issues need to be addressed during the design of the benchmark application. Such as enforcing Cloudstone to benchmark database tier only, enabling the ability of benchmarking replication delay, tweaking time/date function in MySQL for precious resolution to calculate a replication delay, and enforcing clock synchronizations. All the detailed design decisions are discussed as following.

5.1.1 Customized Cloudstone

The Cloudstone benchmark has been designed as a performance measurement tool for Web 2.0 applications. The benchmark mimics a Web 2.0 social events calendar that allows users to perform individual operations such as browsing, searching, and creating events, as well as, social operations such as joining and tagging events (Sobel et al., 2008). Unlike Web 1.0 applications, Web 2.0 applications impose many different behavioral demands on the database. One of the differences is on the write pattern. As contents of Web 2.0 applications depend on user contributions via blogs, photos, videos and tags. More write transactions are expected to be processed. Another difference is on the tolerance with data consistency. In general, Web 2.0 applications are more acceptable to data staleness. For example, it might not be a mission-critical goal for a social network application like Facebook to immediately have a user's new status available to his friends. However, a consistency window of some seconds or even some minutes would be still acceptable. Therefore, it is believed that the design and workload characteristics of the the Cloudstone benchmark is more suitable to the purpose of the study rather than other benchmarks such as TPC-W[2] or RUBiS[3] which are more representative of Web 1.0-like applications.

The original software stack of Cloudstone consists of 3 components: web application, database, and load generator. Throughout the benchmark, the load generator generates load against the web application which in turn makes use of the database. The benchmark has been designed for benchmarking the performance of each tier for Web 2.0 applications. However, the original design of the benchmark limits the purpose of the experiments by mainly focusing on the database tier of the software stack where it is hard to push the database to its performance limit. In general, a user's operation which is sent by a load generator has to be interpreted as database transactions in the web tier based on a predefined business logic before passing the request to the database tier. Thus the saturation on the web tier usually happens earlier than the saturation on the database tier. To prevent this from happening, the design of the original software stack is modified by removing the web server tier. In particular, the business logic of the application is re-implemented in a way that an end-user's operation can be processed directly at the database tier without any intermediate interpretation at the web server tier. Meanwhile, on top of Cloudstone, a DBCP[4] connection pool and a MySQL Connector/J[5] are implemented. The pool component enables the application users to reuse the connections that have been released by other users who have completed their operations in order to save the overhead of creating a new connection for each operation. The proxy component works as a load balancer among the available virtualized database replica servers where all write operations are sent to the master while all read operations are distributed among slaves.

[2]`http://www.tpc.org/tpcw/`

[3]`http://rubis.ow2.org/`

[4]`http://commons.apache.org/dbcp/`

[5]`http://www.mysql.com/products/connector/`

5.1.2 MySQL replication with a fine-grained time/date function

Multiple MySQL replication are deployed to compose the database tier. Two components are implemented to monitor replication delay in MySQL, including a *Heartbeats* database and a time/date function for each virtualized database replica server. The Heartbeats database, synchronized in the form of an SQL statement across replica servers, maintains a *heartbeat* table which records an *id* and a *timestamp* in each row. A heartbeat plug-in for Cloudstone is implemented to periodically insert a new row with a global id and a local timestamp to the master during the experiment. Once the insert query is replicated to slaves, every slave re-executes the query by committing the global id and its own local timestamp. The replication delay from the master to slaves is then calculated as the difference of two timestamps between the master and each slave. In practice, there are two challenges with respect to achieving a fine-grained measurement of replication delay: the resolution of the time/date function and the clock synchronization between the master and slaves. The time/date function offered by MySQL has a resolution of a second which represents an unacceptable solution because accurate measuring of the replication delay requires a higher precision. Thus, a user defined time/date function with a microsecond resolution is implemented based on a proposed solution to MySQL Bug #8523[6]. The clock synchronizations between the master and slaves are maintained by Network Time Protocol (NTP)[7] on Amazon EC2. The system clock is set to synchronize with multiple time servers every second to have a better resolution. More details in dealing with the clock synchronization issue in the cloud will be discussed in Section 5.1.3.

With the customized Cloudstone[8] and the heartbeat plug-in, it is possible to achieve the goal of measuring the end-to-end database throughput and the replication delay. In

[6]`http://bugs.mysql.com/bug.php?id=8523`

[7]`http://www.ntp.org/`

[8]`http://code.google.com/p/clouddb-replication/`

particular, two configurations of the read/write ratios, 50/50 and 80/20 are defined. More over, three configurations of the geographical locations based on *availability zones* and *regions* are also defined and listed as follows where availability zones are defined as distinct locations within a region and zones are separated into geographic areas or countries:

- *Same zone*: all slaves are deployed in the same availability zone of a region of the master database.
- *Different zones*: all slaves are in the same region as the master database, but in different availability zones.
- *Different regions*: all slaves are geographically distributed in a different region from where the master database is located.

The workload and the number of virtualized database replica servers start with a small number and gradually increase at a fixed step. Both numbers stop increasing if there are no throughput gained.

5.1.3 Clock synchronization in cloud

The clock synchronization issue refers to the fact that internal clocks of physical machines may differ due to the initial clock setting and subsequent clock drift. It results in time differences between two machines even though both machines perform the read operation at the same time. This issue could also happen to instances in the cloud environment, if two instances are deployed in distinct physical machines where the clock is not shared. As a matter of fact, it has been observed by Ristenpart et al. (2009) that all instances launched by a single Amazon EC2 account never run in the same physical node. Hence, all running instances that belong to a single account will exhibit the clock synchronization issue.

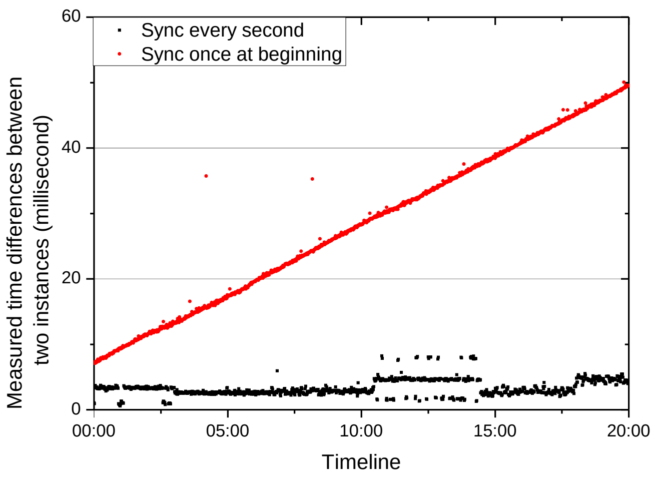


Figure 5.2: Measuring time differences between two instances with and without NTP time synchronization

Figure 5.2 exposes how NTP synchronization keeps the time difference stable between two instances during a 20 minutes period. If two instances only apply the NTP protocol once at the beginning of the experiment, the time difference between two instance surges linearly from 7 milliseconds up to 50 milliseconds. Its median is 28.23, and its standard deviation is 12.31. The surge is caused by clock drift, as Amazon synchronizes its instances in a very relaxed manner - every couple of hours. Thus, clock drift lies in between the two continuous time synchronization. If two instances apply the NTP protocol every second, then samples of all time differences mostly lie between 1 millisecond and 8 milliseconds. Its median is 3.30 ms, and standard deviation is 1.19. With time synchronization enabled every second, the time difference is more stable.

The replication delay in experiments is measured based on committed local timestamps on two or more virtualized database replica servers. Thus, the clock synchronization issue also exists in the replication delay. As the study is more interested in the changes of replication delay, rather than that of accuracy, an average relative replication delay is adopted to eliminate the time differences introduced by the clock synchronization issue. The average relative replication delay is represented as the difference between two average replication delays on the same slave. One average replication delay represents the average of delays without running workloads while another represents the average of delays under a number of concurrent users. Both average is sampled with the top 5% and the bottom 5% data removed as outliers, because of network fluctuation. As both average delays come with stable time differences with NTP protocol enabled every second, the time difference can then be eliminated subtracting the difference. In experiments, the NTP is set to synchronize with multiple time servers every second for a more stable time difference.

5.2 Implementation of benchmark application

As the Figure 5.1 illustrated, the replication experiments are conducted in Amazon EC2. The experiment setup is a three-layer implementation. The Cloudstone benchmark in the first layer controls the read/write ratio and the workload by separately adjusting the number of read and write operations and the number of concurrent users. As a large number of concurrent users emulated by the benchmark could be very resource-consuming, the benchmark is deployed in a large instance to avoid any overload on the application tier. The master database in the second layer receives the write operations from the benchmark and is responsible for propagating the writesets to the slaves. The master database runs in a small instance so that saturation is expected to be observed early. Both the master database server and the application benchmark are deployed in location of *us-east-1a*. The slaves in the third layer are responsible for processing read operations and updating writesets. The number of slaves in a group varies from one to the number where throughput limitation is hit. Several options for the deployment locations of the slaves have been used, namely, the same zone as the master in *us-east-1a*, different zones in *us-east-1b* and four possible different regions, ranging among *us-west*, *eu-west*, *ap-southeast* and *ap-northeast*. All slaves run in small instances for the same reason of provisioning the master instance.

Several sets of experiments have been implemented in order to investigate the end-to-end throughput and the replication delay. Each of these sets is designed to target a specific configuration regarding the geographical locations of the slave databases and the read/write ratio. Multiple runs are conducted by compounding different workloads and numbers of slaves. The benchmark is able to push the database system to a limit where no more throughput can be obtained by increasing the workload and the number of virtualized database replica servers. Every run lasts 35 minutes, including 10 minutes for ramp-up, 20 minutes for steady stage and 5 minutes for ramp-down. Moreover, for each run, both the master and slaves should start with a preloaded, fully-synchronized database.

5.3 Trade-off analysis of virtualized database servers

5.3.1 End-to-end throughput

Figure 5.3 to Figure 5.8 show the throughput trends for up to 4 and 11 slaves with mixed configurations of three locations and two read/write ratios. Both experiment results indicate that MySQL with asynchronous master-slave replication is limited to scale due to the saturation that happened to the master database.

In particular, the throughput trends react to saturation movement and transition in virtualized database replica servers in regard to an increasing workload and an increasing number of replica servers. In general, the observed saturation point (the point right after the observed maximum throughput of a number of slaves), appearing in slaves at the beginning, moves along with an increasing workload when more slaves are synchro-

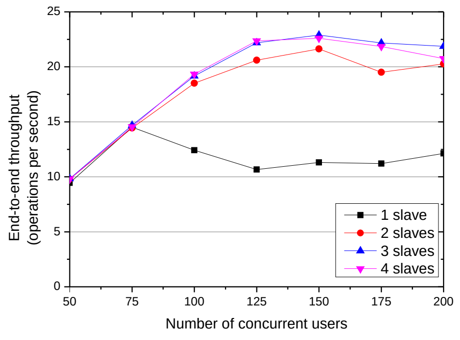


Figure 5.3: End-to-end throughput with 50/50 read/write ratio and 300 initial data size in the same zone

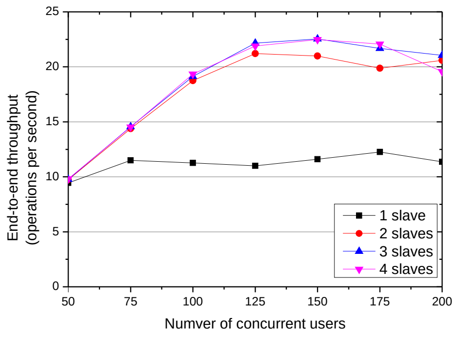


Figure 5.4: End-to-end throughput with 50/50 read/write ratio and 300 initial data size in different zones

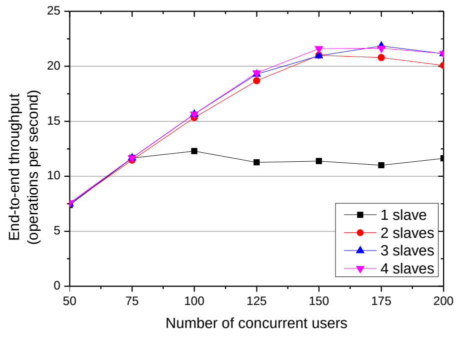


Figure 5.5: End-to-end throughput with 50/50 read/write ratio and 300 initial data size in different regions

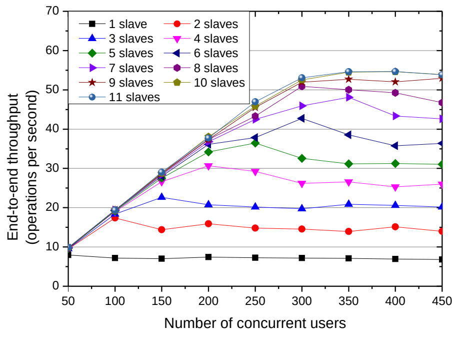


Figure 5.6: End-to-end throughput with 80/20 read/write ratio and 600 initial data size in the same zone

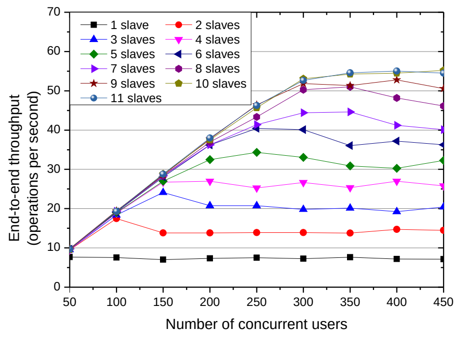


Figure 5.7: End-to-end throughput with 80/20 read/write ratio and 600 initial data size in different zones

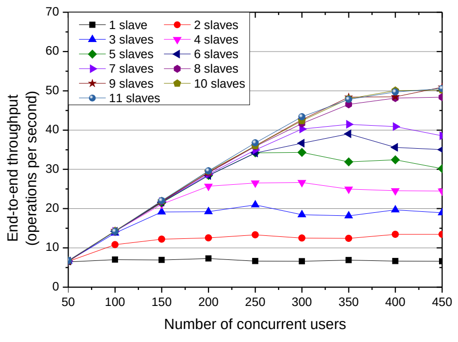


Figure 5.8: End-to-end throughput with 80/20 read/write ratio and 600 initial data size in different regions

nized to the master. But eventually, the saturation will transit from slaves to the master where the scalability limit is achieved. Taking the Figure 5.6 of throughput trends with configurations of same zone and 50/50 ratio as an example, the saturation point of 1 slave is initially observed under 100 workloads due to the full utilization of the slave's CPU. When a 2nd slave is attached, the saturation point shifts to 175 workloads where both slaves reach their maximum CPU utilization while the master's CPU usage rate is also approaching its limit. Thus, ever since the 3rd slave is added, 175 workloads remain as the saturation point, but with the master being saturated instead of the slaves. Once the master is in the saturation status, adding more slaves does not help with improving the scalability because the overloaded master fails to offer extra capacity for improving write throughput to maintain the read/write ratio that corresponds to the increment of the read throughput. Hence, the read throughput is constrained by the benchmark, for the purpose of maintaining the predefined read/write ratio at 50/50. The slaves are over provisioned in the case of 3 and 4 slaves, as the suppressed read throughput prevents slaves from being fully utilized. The similar saturation transition also happens to 3 slaves at 50/50 ratio in different zones and different regions in Figure 5.4 and Figure 5.5 respectively, 10 slaves at 80/20 ratio in the same zone and different zones in Figure 5.6 and Figure 5.7 respectively, and also 9 slaves at 80/20 ratio in different regions in Figure 5.8.

The configuration of the geographic locations is a factor that affects the end-to-end throughput, in the context of locations of users. In the case of our experiments, since all users emulated by Cloudstone send read operations from *us-east-1a*, distances between the users and the slaves increase by following in the order of same zone, different zones and different regions. Normally, a long distance incurs a slow round-trip time, which results in a small throughput for the same workload. Therefore, it is expected that a decrease of maximum throughput can be observed when configurations of locations follow the order of same zone, different zones and different regions. Moreover, the throughput degradation is also related to read percentages, the higher percentage the larger degradation. It explains why degradation of maximum throughput is more significant with the configuration of 80/20 read/write ratio as shown in Figure 5.6 to Figure 5.8. Hence, it is a good strategy to distribute replicated slaves to places that are close to users to improve end-to-end throughput.

The performance variation of instances is another factor that needs to be considered when deploying a database in the cloud. For throughput trends of 1 slave at 50/50 read-/write ratio with configurations of different zones and different regions, respectively, if the configuration of locations is the only factor, it is expected that the maximum throughput in different zones in Figure 5.4 would be larger than the one in different regions in Figure 5.5. However, the main reason of throughput difference here is caused by the performance variation of instances rather than the configuration of the locations. The 1st slave from the same zone runs on top of a physical machine with an Intel Xeon E5430 2.66GHz CPU. While another 1st slave from different zones is deployed in a physical machine powered by an Intel Xeon E5507 2.27GHz CPU. Because of the performance differences between physical CPUs, the slave from the same zone performs better than the one from different zones. Previous research indicated that the coefficient of variation of CPU of small instances is 21% (Schad et al., 2010). Therefore, it is a good strategy to validate the instance performance before deploying applications into the cloud, as poor-performing instances are launched randomly and can largely affect application performance.

5.3.2 Replication delay

Figure 5.9 to Figure 5.14 show the trends of the average relative replication delay for up to 4 and 11 slaves with mixed configurations of three locations and two read/write ratios. The results of both figures imply that the impact of the configurations of the

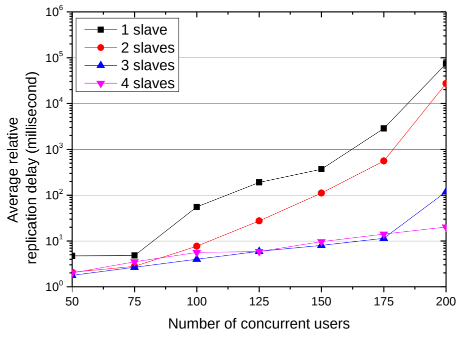


Figure 5.9: Average relative replication delay with 50/50 read/write ratio and 300 initial data size in the same zone

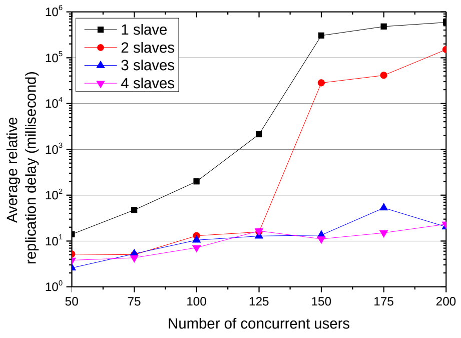


Figure 5.10: Average relative replication delay with 50/50 read/write ratio and 300 initial data size in different zones

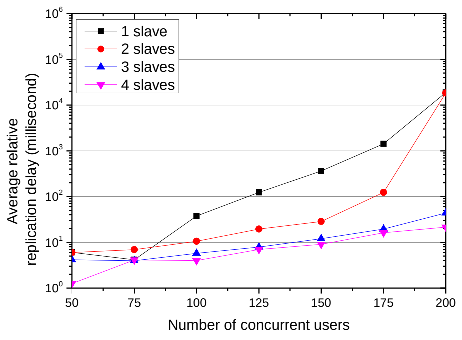


Figure 5.11: Average relative replication delay with 50/50 read/write ratio and 300 initial data size in different regions

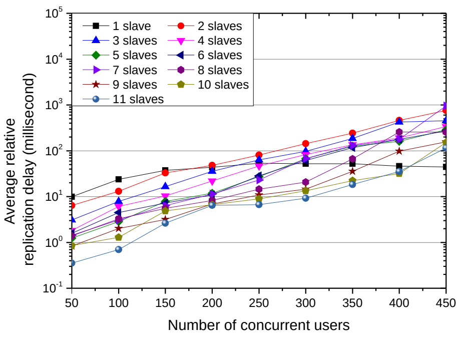


Figure 5.12: Average relative replication delay with 80/20 read/write ratio and 600 initial data size in the same zone

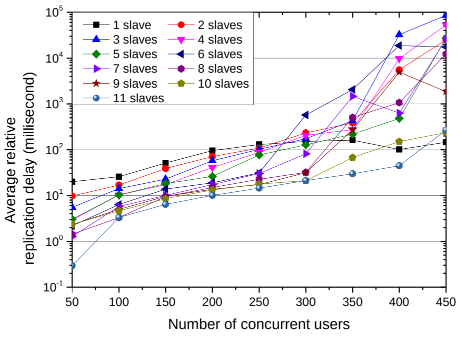


Figure 5.13: Average relative replication delay with 80/20 read/write ratio and 600 initial data size in different zones

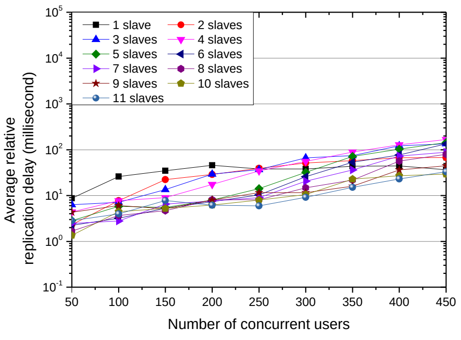


Figure 5.14: Average relative replication delay with 80/20 read/write ratio and 600 initial data size in different regions

geographical locations on replication delay is less important than that from the workload characteristics. The trends of the average relative replication delay respond to an increasing workload and an increasing number of virtualized database replica servers.

For most cases, with the number of virtualized database replica servers being kept constant, the average relative replication delay surges along with an increasing workload. Because an increasing workload leads to more read and write operations sent to the slaves and the master database, respectively, the increasing read operations result in a higher resource demand on every slave, while the increasing write operations on the master database leads to, indirectly, increasing resource demand on slaves as more writesets are propagated to be committed on slaves. The two increasing demands push resource contention higher, resulting in the delay of committing writesets which subsequently increasing replication delay. Similarly, the average relative replication delay decreases along with an increasing number of replica servers as adding a new slave leads to a reduction in the resource contention and hence decreasing the replication delay.

The configuration of the geographic location of the slaves play a less significant role in affecting replication delay, in comparison to the change of the workload characteristics. We measured the 1/2 round-trip time between the master in *us-west-1a* and the slave that uses different configurations of geographic locations by running *ping*[9] command every second for a 20-minute period. The results suggest an average of 16, 21, and 173 milliseconds for the 1/2 round-trip time for the same zone in Figures 5.9 and 5.12, different zones in Figure 5.10 and 5.13, and different regions in Figures 5.11 and 5.14, respectively. However, the trends of the average relative replication delay can usually go up to two to four orders of magnitude as shown from Figures 5.9 to 5.11, or one to three orders of magnitude as shown in Figures 5.12 to 5.14. Therefore, it could be suggested that the geographic replication would be applicable in the cloud as long as workload

[9]`http://linux.die.net/man/8/ping`

characteristics can be well managed, such as having a smart load balancer which is able to balance the operations based on the estimated processing time.

5.4 Discussion

In practice, there are different approaches for deploying data-intensive applications in cloud platforms. In this chapter, the study is focused on the *virtualized database servers* approach where the resources of the database tiers are migrated to virtual machines in the public cloud. The behavior of the master-slave database replication strategy on Amazon EC2 has been experimentally evaluated using the Cloudstone benchmark and MySQL databases. The experiments involved two configurations of different workload read/write ratios, namely 50/50 and 80/20, and different configuration of the geographical locations of the virtualized database replica servers.

The results of the study show that the performance variation of the dynamically allocated virtual machines is an inevitable issue that needs to be considered when deploying database in the cloud. Clearly, it affects the end-to-end throughput. Additionally, different configurations of geographic locations can also noticeably affect the end-to-end throughput. For most cases, as the number of workload increases, the replication delay increases. However, as the number of slaves increases, the replication delay is found decreases. The effect of the configurations of geographic location is not as significant as increasing workloads in affecting the replication delay.

Chapter 6

A framework of SLA-driven database replication on virtualized database servers

One of the main advantages of the cloud computing paradigm is that it simplifies the time-consuming processes of hardware provisioning, hardware purchasing and software deployment. Currently, the increasing numbers of cloud-hosted applications are generating and consuming increasing volumes of data at an unprecedented scale. Cloud-hosted database systems, such as virtualized database servers, powering these applications form a critical component in the software stack of these applications. Service level agreements (SLAs) represent the contract which captures the agreed upon guarantees between a service provider and its customers. The specifications of existing SLA for cloud services are not designed to flexibly handle even relatively straightforward performance and technical requirements of customer applications.

In this chapter, the problem of adaptive customer-centric management for replicated virtualized database servers in single or multiple data centers is tackled. A novel adaptive approach for SLA-based management of virtualized database servers from the customer perspective is presented. The framework is database platform-agnostic, supports virtualized database servers, and requires zero source code changes of the cloud-hosted software applications. It facilitates dynamic provisioning of the database tier in software stacks based on application-defined policies for satisfying their own SLA performance requirements, avoiding the cost of any SLA violation and controlling the monetary cost of the allocated computing resources. In this framework, the SLA of the customer applications are declaratively defined in terms of goals which are subjected to a number of constraints that are specific to the application requirements. The framework continuously monitors the application-defined SLA and automatically triggers the execution of necessary corrective actions, such as scaling out the database tier, when required. Therefore, the framework is able to keep several virtualized database replica servers in different data centers to support the different availability, scalability and performance improvement goals. The experimental results demonstrate the effectiveness of the SLA-based framework in providing the customer applications with the required flexibility for achieving their SLA requirements.

The remainder of this chapter is structured as follows. Section 6.1 introduces the architecture of the adaptive framework. Details of the experiment implementation of the different components of the framework are discussed on Section 6.2. Then, the results of the experimental evaluation for the performance of the approach are presented in Section 6.3, followed by discussions and conclusions in Section 6.4.

6.1 Architecture of SLA management framework

Figure 6.1 shows an overview of the framework architecture which consists of three main modules: the *monitor module*, the *control module* and the *action module*. In this architecture, the monitor module is responsible for continuously tracking the replication delay of each virtualized database replica server and feeding the control module with the collected information. The control module is responsible for continuously checking the replication delay of each replica server against its associated application-defined SLA of data freshness and triggers the action module to scale out the database tier with a new virtualized database replica server when it detects any SLA-violation in any current replica server.

The key design principles of the framework architecture are to be application-independent and to require no code modification on the customer software applications that the framework will support. In order to achieve these goals, the framework relies on a database proxying mechanism which forwards database requests to the underlying databases and returns the results to the client transparently using an intermediate piece of software, the proxy, without the need of having any database drivers installed (Sakr and Liu, 2012). In particular, a database proxy software is a simple program that sits between the client application and the database server that can monitor, analyze or transform their communications. Such flexibility allows for a wide variety of uses such as load balancing, query analysis and query filtering. The implementation details for each of the three main modules of the framework architecture will be discussed in the remaining part of the section.

As mentioned before, the design of the framework follows two main principles, function-extensible and application-independent. Any new objectives, such as throughput, can be easily added with pairs of implementations in both monitor and control modules. Actions, such as starting a new virtualized database replica server, for new objectives can be reused from a list of available actions in the action module, or can be added when no satisfied actions is found. It is worth bearing in mind that all objectives are added with no code modification to existing application that is managed by the frame-

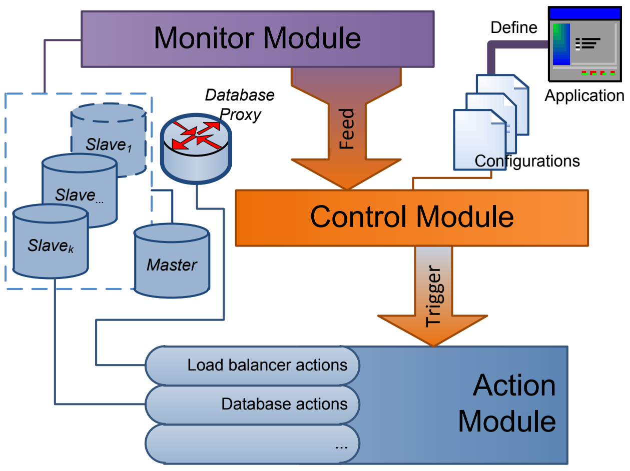


Figure 6.1: The SLA management framework architecture

work. However, some tools, databases, or plug-ins need to be enabled at the system level to enable the objectives to be monitored properly, for example, recording all queries to be bypassed in the load balancer.

In this chapter, the study specifically focuses on the implementation of replication and consistency management of the SLAs management framework. Meanwhile, the independence of the framework will be demonstrated by integrating the framework with a database-focused Cloudstone implementation. The tools, databases, and plug-ins that are enabled for proper monitoring are addressed in Section 6.2.

6.1.1 Monitor module

The monitor module is responsible for tracking the replication delay between the virtualized database master server and each virtualized database replica server. The replication delay for each replica server is computed by measuring the time difference of two associated local timestamps committed on the master and the replica server. Therefore, a *Heartbeats* database is created in the master and each synchronized slave database server. Each Heartbeats database maintains a *heartbeat* table with two fields: an *id* and a *timestamp*. A database request to insert a new record with a global id and a local timestamp is periodically sent to the master. Once the insert record request is replicated to the slaves, every slave re-executes the request by committing the same global id and its own local timestamp. The update frequency of a record in the master is configurable, named as *heartbeat interval* in millisecond unit. The default configuration of the heartbeat interval is set to be 1 second in the experiments. While records are updated in the master database and propagated over all slaves periodically, the monitor module maintains a pool of threads that are run frequently to read up-to-date records from the master and slaves. The read frequency is also a configurable parameter in millisecond unit, known as *monitor interval*. In order to reduce the burden of repetitive read requests on the virtualized database replica servers, all records are only fetched once, and all local timestamps extracted from records are kept locally in the monitor module for further calculation.

The replication delay calculation between the master and a slave is initiated by the corresponding thread of the slave every time after fetching the records. In the general case of assuming that there are n and k local timestamps in total in the master array, $timestamps_m$, and the slave array, $timestamps_s$, the slave's ith replication delay $delay[i]$ is computed as follows:

$$delay[i] = timestamps_s[i] - timestamps_m[i] \tag{6.1}$$

where $i \leq k = n$ and the master and the slave databases are fully synchronized. In the case of $k < n$ where there is partial synchronization between the master and the slave databases which composes of both a consistent part and an inconsistent part, the computation of the $delay[i]$ of the slave can be broken into two parts: The delay of the consistent part with $i \leq k$ is computed using Equation 6.1.

The delay of the inconsistent part with $k < i \leq n$ is computed as follows:

$$\begin{aligned} delay[i] &= timestamps_s[k] - timestamps_m[k] \\ &\quad + timestamps_m[i] - timestamps_m[k] \end{aligned} \tag{6.2}$$

In the case of $n < k$ where indeterminacy could happen due to the missing of $k + 1^{\text{th}}$ local timestamp and beyond (this situation could happen when a recent fetch of the slave occurs later than the fetch of the master), the $delay[i]$ of the slave uses Equation 6.1 for $i \leq n$ and the $delay[i]$ of the slave for $n < i \leq k$ will be neglected as there is no appropriate local timestamps of the master that can be used for calculating the replication delay. The neglected calculations will be carried out later after the array of the master is updated.

6.1.2 Control module

The control module maintains the configuration information about:

- The configurations of the load balancer, including proxy address and proxy script.
- The configurations of the monitor module, such as heartbeat interval and monitor interval.
- The access information of each virtualized database replica server, namely host address, port number, user name, and password.
- The location information of each virtualized database replica server, such as *us-east*, *us-west*, *eu-west*.
- And in addition to the application-defined SLA, the tolerated replication delay of each virtualized database replica server for this study.

In practice, the SLA of the replication delay for each virtualized database replica server, $delay_{sla}$, is defined as an integer value in the unit of millisecond which represents two main components:

$$delay_{sla} = delay_{rtt} + delay_{tolerance} \tag{6.3}$$

where the round-trip time component of the SLA replication delay, $delay_{rtt}$, is the average round-trip time from the virtualized database master server to the virtualized database replica server. In particular, it represents the minimum delay cost for replicating data from the master to the associated slave. The tolerance component of the replication delay, $delay_{tolerance}$, is defined by a constant value which represents the tolerance limit of the period of the time for the replica server to be inconsistent. This tolerance component can vary from one replica server to another depending on many factors such as the application requirements, the geographic location of the replica server, and the workload characteristics and the load balancing strategy of each application.

One of the main responsibilities of the control module is to trigger the action module for adding a new virtualized database replica server, when necessary, in order to avoid any violation in the application-defined SLA of data freshness for the active replicas. In framework implementation, an intuitive strategy is followed to trigger the action module for adding a new replica server when it detects a number of continuous up-to-date monitored replication delays of a replica server which exceeds its application-defined threshold, T, of SLA violation of data freshness. In other words, for a running replica server, if the latest T monitored replication delays are violating its SLA of data freshness, the control module will trigger the action module to activate the geographically closest replica server according to the location of the violating replica server. It is worthy to note that the strategy of the control module in making the decisions regarding the addition a new replica server in order to avoid any violence of the application-defined SLA can play an important role in determining the overall performance of the framework. However, it is not the main focus of this paper to investigate different strategies for making these decisions. This aspect will be left for future work.

In the last chapter, it has been noted that the effect of the configurations of geographic location of the virtualized database replica server is not as significant as the effect of the overloading workloads in increasing the staleness window of the replica servers. Therefore, the control module can decide to stop an active replica server when it detects a decreasing workload that can be served by less number of replica servers without violating the application-defined SLAs in order to reduce the monetary cost of the running application.

6.1.3 Action module

The action module is responsible for adding a new virtualized database replica server when it is triggered by the action module. In general, adding a new replica server involves extracting database content from an existing replica server and copying that content to a new replica server. In practice, the time of executing these operations mainly depends on the database size. To provision virtualized database replica servers in a timely fashion, it is necessary to periodically snapshot the database state in order to minimize the database extraction and copying time to that of only the snapshot synchronization time. There is a trade-off between the time to snapshot the database, the size of the transactional log and the amount of update transactions in the workload. This trade-off can be further optimized by applying recently proposed live database migration techniques (Cecchet et al., 2011; Elmore et al., 2011).

In order to keep the experiments focused on the main concerns of the framework, a set of hot backups, which are originally not used for serving the application requests but kept synchronized, are used and then can be made active and used by the load balancer for serving the application requests when the action module is triggered for adding a new virtualized database replica server. The study of the cost and effect of the live database migration activities will also be left as future work.

6.2 Implementation of SLA management framework

Figure 6.2 illustrates the setup of experiments for the SLA management framework in the Amazon EC2 platform. Besides the SLA management framework, the experiment setup also adopts the customized Cloudstone benchmark from Section 5.1.1, MySQL replication with a fine-grained time/date function from Section 5.1.2, and MySQL Proxy[1], as necessary, components.

The experiment setup is a multiple-layer implementation. The first layer represents the Cloudstone benchmark which generates an increasing workload of database requests

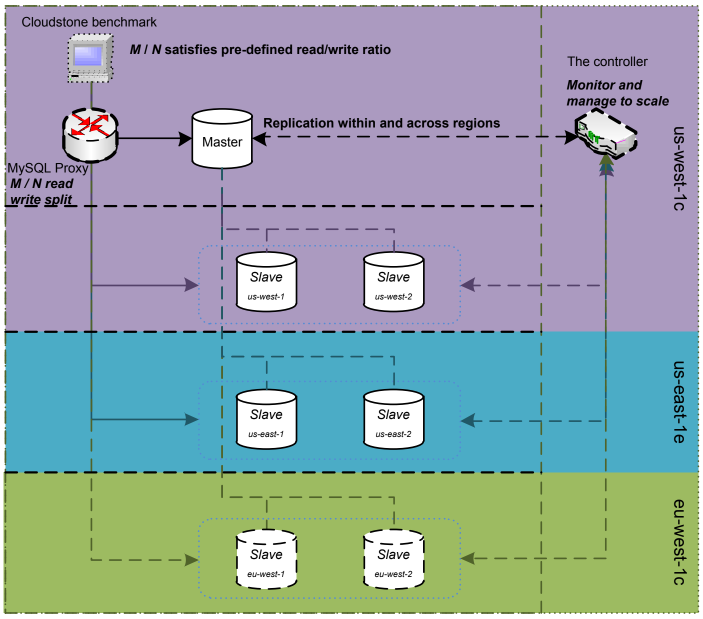


Figure 6.2: The implementation of the SLA management framework in the setup of experiments

[1]https://launchpad.net/mysql-proxy

with a fixed read/write ratio. The benchmark is deployed in a large instance to avoid any overload on the application tier. The second layer hosts the MySQL Proxy and the SLA management framework. MySQL Proxy with read and write split enabled resides in the middle between the benchmark and the virtualized database replica servers, and acts as a load balancer to forward read and write operations to the master and slaves correspondingly. The third layer represents the database tier that consists of all the replica servers where the master database receives the write operations from the load balancer after which it becomes responsible for propagating the *writesets* to all the virtualized database slave servers. The master database runs in a small instance so that an increasing replication delay is expected to be observed along with an increasing workload. The master database is closely located to the benchmark, the load balancer and the SLA management framework. They are all deployed in the location of *us-west*. The slave servers are responsible for serving the read operations and updating the writesets. They are deployed in three regions, namely: *us-west*, *us-east* and *eu-west*. All slaves run in small instances for the same reason of provisioning the master instance.

Two sets of experiments are implemented in order to evaluate the effectiveness of the SLA management framework in terms of its effectiveness on maximizing the end-to-end system throughput and minimizing the replication delay for the underlying virtualized database servers. In the first set of experiments, the value of the tolerance component, $delay_{tolerance}$, of the SLA replication delay is fixed at 1000 milliseconds, and the monitor interval, $intvl_{mon}$, is varied among the following set of values, 60, 120, 240, and 480 seconds. In the second set of experiments, in contrast to the first test, the monitor interval, $intvl_{mon}$, is fixed at 120 seconds, and the SLA of replication delay is adjusted by varying the tolerance component of the replication delay, $delay_{tolerance}$, among the values of 500, 1000, 2000, and 4000 milliseconds. In the experiment environment, the round-trip component for the virtualized database replica servers is determined with *ping* command running every second for a 10 minutes period. The average round-trip time of three geographical regions is 30, 130, and 200 milliseconds from the master to slaves in *us-west*, *us-east*, and *eu-west* respectively.

Every experiment runs for a period of 3000 seconds with a starting workload of 220 concurrent users and database requests with read/write ratio at 80/20. The workload gradually increases in steps of 20 concurrent users every 600 seconds so that each experiment ends with a workload of 300 concurrent users. Each experiment deploys 6 virtualized database replica servers in 3 regions where each region hosts two replica servers: the first replica server is an active replica which is used from the start of the experiment for serving the database requests of the application while the second one is a hot backup which is not used for serving the application requests at the beginning of the experiment but can be added by the action module, as necessary, when triggered by the control module. Finally, in addition to the two sets of experiments, two experiments without the adaptive SLA management framework are conducted as baselines for measuring the end-to-end throughputs and replication delays of 3 and 6 slaves, representing the minimum and the maximum number of running replica servers, respectively. For all experiments, the value of the heartbeat interval, $intvl_{heart}$, is set to 1 second and the value of the threshold, T, for the maximum possible continuous SLA violations for any replica server is calculated using the following formula $T = \frac{intvl_{mon}}{intvl_{heart}}$.

6.3 Evaluation of SLA management framework

6.3.1 End-to-end throughput

Table 6.1 presents the end-to-end throughput results for the set of experiment with different configuration parameters. The baseline experiments represent both the minimum and the maximum end-to-end throughput results with 22.33 and 38.96 operations per second respectively. The end-to-end throughput delivered by the adaptive SLA management framework for the different experiments fall between the two baselines based on the variance on the monitor interval, $intvl_{mon}$, and the tolerance of replication delay, $delay_{tolerance}$. However, it is worth noting that the end-to-end throughput can be still affected by a lot of performance variations in the cloud environment such as hardware performance variation, network variation and warm up time of the virtualized database servers. Similarly, The two baseline experiments also represent the minimum and the maximum running time of all virutalized database replica servers with 9000 and $18{,}000$ seconds respectively. Therefore, the total running time of the replica servers for the different experiments fall within the range of 9000 and $18{,}000$ seconds. Each experiment starts with 3 active replicas which are gradually increased during the experiments based on the configurations of the monitor interval and the SLA of replication delay parameters until it finally ends with 6 replica servers.

In general, the relationship between the running time of all slaves and end-to-end throughput is not straightforward. Intuitively, a longer monitor interval or a longer tolerance of replication delay usually postpones the addition of new virtualized database replica servers and consequently reduces the end-to-end throughput. The results show that the tolerance of the replication delay parameter, $delay_{tolerance}$ is more sensitive than the monitor interval parameter, $intvl_{mon}$. For example, setting the values of the tolerance of the replication delay to 4000 and 1000 result in longer running times of the replica servers than when the values are set to 2000 and 500. On the other hand, the increase of running time of all replica servers clearly follows a linear trend along with the increase of the end-to-end throughput. However, a general conclusion can not be made as the trend is likely affected by the workload characteristics.

Table 6.1: The effect of the the adaptive SLA management framework on the end-to-end system throughput

Experiment Parameters	The monitor interval, $intvl_{mon}$, in seconds	The tolerance of replication delay, $delay_{tolerance}$, in milliseconds	Number of running replica servers	Running time of all replica servers in seconds	End-to-end throughput in operations per seconds	Figure
Baselines with fixed number of replica servers	N/A	N/A	3	9000	22.33	Figure 6.3
	N/A	N/A	6	18,000	38.96	Figure 6.4
Varying the monitor interval, $intvl_{mon}$	60	1000	$3 \rightarrow 6$	15,837	38.43	Figure 6.5
	120	1000	$3 \rightarrow 6$	15,498	36.45	Figure 6.6
	240	1000	$3 \rightarrow 6$	13,935	34.12	Figure 6.7
	480	1000	$3 \rightarrow 6$	12,294	31.40	Figure 6.8
Varying the tolerance of replication delay, $delay_{tolerance}$	120	500	$3 \rightarrow 6$	15,253	37.44	Figure 6.9
	120	1000	$3 \rightarrow 6$	15,498	36.45	Figure 6.6
	120	2000	$3 \rightarrow 6$	13,928	36.33	Figure 6.10
	120	4000	$3 \rightarrow 6$	14,437	34.68	Figure 6.11

6.3.2 Replication delay

Figures 6.3 to 6.11 illustrate the effect of the adaptive SLA management framework on the performance of the replication delay for the virtualized database replica servers. Figures 6.3 and 6.4 show the replication delay of the two baseline cases that will be used for comparison purposes. They represent the experiments of running with a fixed number of virtualized database replica servers, 3 and 6 respectively, from the start until the end of the experiments. Figure 6.3 shows that the replication delay tends to follow different patterns for different replica servers. The two trends of virtualized database servers in *us-west-1* and *eu-west-1* surge significantly at 260 and 280 users respectively. At the same time, the trend of virtualized database server in *us-east-1* tends to be stable through out the entire running time of the experiment. The main reason behind that is the performance variation between the virtualized database servers for replicas, as both virtualized database servers in *us-west-1* and *eu-west-1* are powered by Intel(R) Xeon(R) E5507 @ 2.27GHz CPU, whereas the server in *us-east-1* is deployed with a higher performance CPU, Intel(R) Xeon(R) E5645 @ 2.40GHz CPU. Due to the performance differences between the physical CPUs specifications, the virtualized database server in *us-east-1* is able to handle the amount of operations that saturated the servers in *us-west-1* and

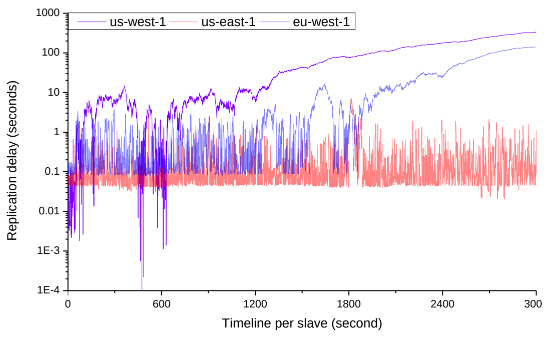


Figure 6.3: The performance of the replication delay for 3 replica servers with the framework disabled

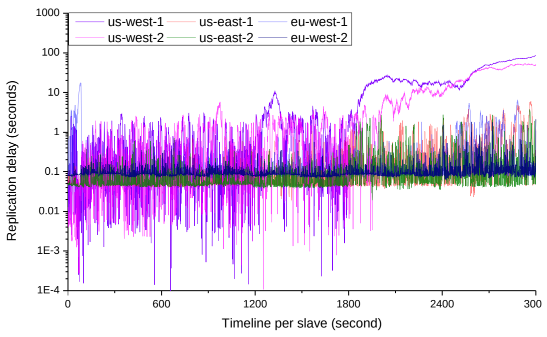


Figure 6.4: The performance of the replication delay for 6 replica servers with the framework disabled

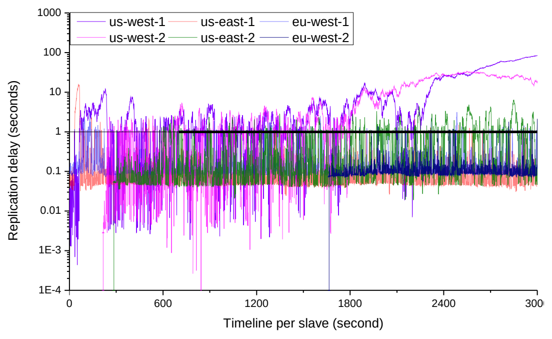


Figure 6.5: The performance of the replication delay for up to 6 replica servers with the framework enabled, $delay_{tolerance} = 1000$ milliseconds, and $intvl_{mon} = 60$ seconds

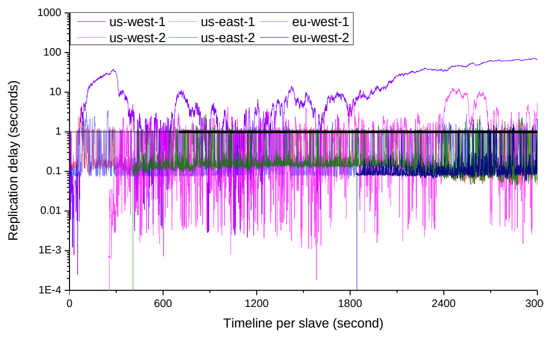


Figure 6.6: The performance of the replication delay for up to 6 replica servers with the framework enabled, $delay_{tolerance} = 1000$ milliseconds, and $intvl_{mon} = 120$ seconds

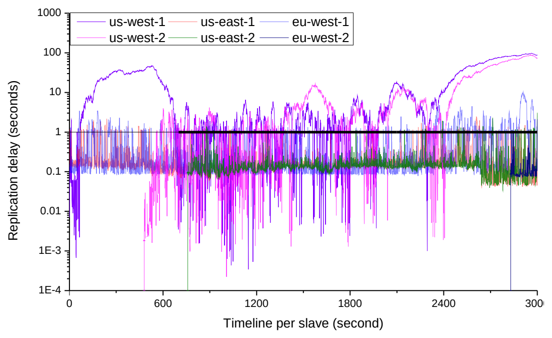


Figure 6.7: The performance of the replication delay for up to 6 replica servers with the framework enabled, $delay_{tolerance} = 1000$ milliseconds, and $intvl_{mon} = 240$ seconds

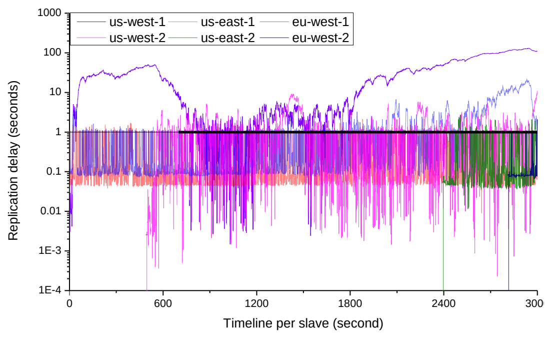


Figure 6.8: The performance of the replication delay for up to 6 replica servers with the framework enabled, $delay_{tolerance} = 1000$ milliseconds, and $intvl_{mon} = 480$ seconds

eu-west-1. Moreover, with an identical CPU for *us-west-1* and *eu-west-1*, the former seems to surge at an earlier point than the latter. This is basically because of the difference in the geographical location of the two virtualized database servers. As illustrated in Figure 6.2, the MySQL Proxy location is closer to the virtualized database server in *us-west-1* than the server in *eu-west-1*. Therefore, the forwarded database operations by the MySQL Proxy take less time to reach the server in *us-west-1* than to the server in

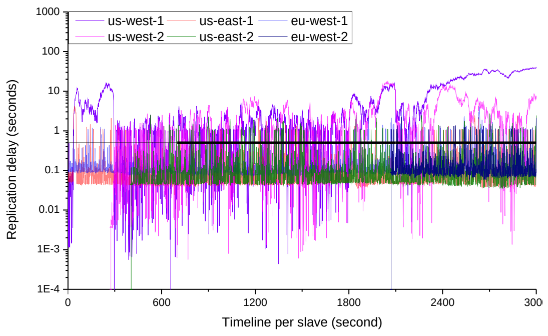


Figure 6.9: The performance of the replication delay for up to 6 replica servers with the framework enabled, $delay_{tolerance} = 500$ milliseconds, and $intvl_{mon} = 120$ seconds

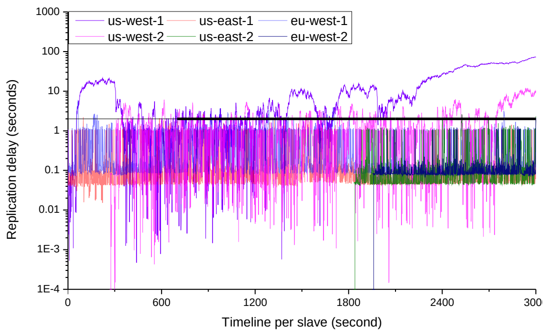


Figure 6.10: The performance of the replication delay for up to 6 replica servers with the framework enabled, $delay_{tolerance} = 2000$ milliseconds, and $intvl_{mon} = 120$ seconds

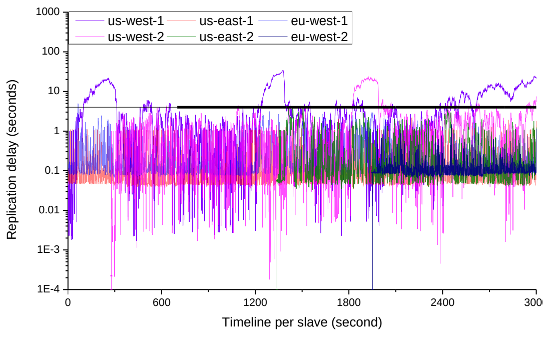


Figure 6.11: The performance of the replication delay for up to 6 replica servers with the framework enabled, $delay_{tolerance} = 4000$ milliseconds, and $intvl_{mon} = 120$ seconds

eu-west-1 which leads to more congestion on the *us-west-1* side. Similarly, in Figure 6.4, the replication delay tends to surge in both virtualized database servers in *us-west-1* and *us-west-2* for the same reason of the difference in the geographic location of the underlying virtualized database server.

Figure 6.6, and Figures 6.9 to 6.11 show the results of the replication delay for the experiments using different values for the monitor interval, $intvl_{mon}$, and the tolerance of replication delay, $delay_{tolerance}$, parameters. For example, Figure 6.6 shows that the virtualized database replica servers in *us-west-2*, *us-east-2*, and *eu-west-2* are added in sequence at the 255th, 407th, and 1843th seconds, where the drop lines are emphasized. The addition of the three replica servers are caused by the SLA-violation of the virtualized database replica server in *us-west-1* at different periods. In particular, there are four SLA-violation periods for the replica server in *us-west-1* where the period must exceed the monitor interval, and all calculated replication delays in the period must exceed the SLA of replication delay. These four periods are: from 67 to 415 in total of 349 seconds, from 670 to 841 for a total of 172 seconds, from 1373 to 1579 for a total of 207 seconds, and from 1615 to 3000 for a total of 1386 seconds. The adding of new replica servers is only triggered on the 1st and the 4th periods based on the time point analysis. The 2nd and the 3rd periods do not trigger the addition of any new replica servers as the number of detected SLA violations does not exceed the defined threshold, T.

Figures 6.5 to 6.8 show the effect of varying the monitor interval, $intvl_{mon}$ on the replication delay of the different virtualized database replica servers. The results show that virtualized database replica server in *us-west-2* is always the first location that add a new replica server because it is the closest location to the virtualized database server in *us-west-1* which hosts the replica server that is first to violate its defined SLA data freshness. The results also show that as the monitor interval increases, the triggering points for adding new replica servers are usually delayed. On the contrary, the results of Figure 6.6 and Figures 6.9 to 6.11 show that increasing the value of the tolerance of the replication delay parameter, $delay_{tolerance}$, does not necessarily cause a delay in the triggering point for adding new replica servers.

6.4 Discussion

In general, the results of experiments show that the adaptive SLA management framework can play an effective role on reducing the replication delay of the underlying virtualized database servers by adding new virtualized database replica servers when necessary. It is also observed that with more replica servers added, the replication delay for the overloaded replicas can dramatically drop. Moreover, it is more cost-effective in comparison to the over-provisioning approach for the number of virtualized database replica servers that can ensure low replication delay because it adds new replica servers only when necessary based on the application-defined SLA of data freshness for the different underlying virtualized database servers.

This chapter presented an adaptive SLA management framework for replicating virtualized database servers. The framework provides the software applications with flexible mechanisms for maintaining several replica servers in different data centers with different levels of SLAs for their data freshness.

Chapter 7

Conclusion and future work

7.1 Conclusion

This thesis has presented two generic performance evaluation solutions for different cloud platforms and cloud databases, as well as proposing an end-to-end framework for customer-centric SLA management of virtualized database servers in two steps, with performance evaluation of database replication of virtualized database servers in the first step with the following step being that of by building, implementing, and evaluating such a framework.

A general framework for performance evaluation of cloud platforms

A novel architecture runtime evaluation framework for cloud platforms is introduced in Chapter 3. The framework includes a number of prebuilt, preconfigured, and reconfigurable components for conducting cloud performance evaluations across a number of example target platforms. The framework is tailored for evaluating various aspects of a cloud platform at runtime. Given the different characteristics of different cloud platforms, the unified interface in the framework allows direct comparison of different cloud platforms where was simply not possible before. Empirical results show the framework is a feasible approach and can be used to identify areas of performance bottlenecks that have a significant impact on a platform's runtime performance.

Performance evaluation of database replication of NoSQL database as a service

A generic performance evaluation architecture is presented in Chapter 4 to examine the customers' experience of NoSQL database as a service, in regard to weak consistency and possible performance trade-offs to justify its use. The study found that platforms differed widely in how much weak consistency is observed by customers. On some platforms, the customer did not observe any inconsistency over several million reads through a week. While inconsistency is, in theory, possible, its occurrence has been very rare; perhaps only happening if there is a failure of one of the nodes or communication links used in the computation. It has been observed that the customer who requests eventual consistent reads on the Amazon SimpleDB platform experiences frequent stale reads and inter-item inconsistency. Also, this choice does not provide other desirable properties like read-your-writes and monotonic reads. The output of this study should help customers who are seeking to understand the properties of the new NoSQL storage platforms for the cloud, and who need to make sensible choices about which storage platform to use. Any system design needs to have recourse to manual processes to fix the mistakes and errors that occur due to the inherent limitations of the platform. The same manual process should be sufficient to mitigate the risk of inconsistency-induced difficulties especially when great care has been taken in the design of the application to address the dangers of eventual consistency.

Performance evaluation of database replication of virtualized database servers

In Chapter 5, the performance evaluation of database replication is focused on the virtualized database servers where the resources of the database tiers from conventional data centers are migrated to virtual machines in the public cloud. The behavior of the master-slave database replication strategy on Amazon EC2 is examined through sets of experiments using a customized Cloudstone benchmark. The experiments involved two configurations of the workload read/write ratio at 50/50 and 80/20, and different configurations of the geographical locations of virtualized database replicas servers. The results show that the performance variation of the dynamically allocated virtual machines is an inevitable issue that needs to be considered when deploying virtualized database servers in the cloud as it affects the end-to-end throughput. Meanwhile, different configurations of geographic locations has a noticeable impact on the end-to-end throughput. For most cases, as the number of workload increases, the replication delay increases. However, as the number of slaves increases, the replication delay decreases. The effect of the configurations of geographic location on the replication delay is not as significant as that which is caused by increasing workloads. This means that a geographic distributed database systems is applicable for use with virtualized database servers.

A framework of SLA-driven database replication on virtualized database servers

Chapter 6 takes the observations from the performance evaluation of database replication of virtualized database servers to propose the design and implementation of an end-to-end framework that facilitates adaptive and dynamic provisioning of the database tier of the software applications based on customer-centric policies for satisfying their own SLA performance requirements by avoiding the cost of any SLA violation and controlling the monetary cost of the allocated computing resources. The framework provides the customer applications with declarative and flexible mechanisms for defining their specific requirements for fine-grained SLA metrics at the application level. The framework is database platform-agnostic, uses virtualized database servers and requires zero source code changes of the cloud-hosted software applications.

7.2 Future work

Because of conflicting nature between the service level agreements (SLAs) that the cloud customers want to offer to their end-users and the existing SLAs supported by cloud providers. Most providers guarantee only the availability of their services (Suleiman et al., 2012). Automatic SLA management framework for handling customers' cloud databases, especially for those of virtualized database servers, is expected to grow with increasing requirements and importance. This leads to a number of interesting problems for future research:

Data partition

In the case of an overload detection on virtualized database servers, a simple scaling out strategy is to replicate the whole database. However, such an action is coarse-grained, because migrating a whole database usually takes a long time, sometimes even longer than the period of the overload. It is also reported that spikes leading to an overload usually happen on certain data partitions, rather than the whole database (Bodik et al., 2010). Therefore, if these overload-prone data partitions can be detected and replicated, the scaling out action would be more efficient with less migration time.

Live migration

A simple migration process involves flushing logs onto the disk, taking down the running database, migrating data to another location, and bringing up the migrated database. There is a significant downtime between taking down the running database and bringing up the migrated database. Live migration is an approach towards less or zero downtime. There have been a few interesting solutions for general operating systems in virtual machines (Clark et al., 2005), shared nothing databases (Elmore et al., 2011), and shared storage databases (Das et al., 2011). It may be challenging and interesting to apply similar approaches into the virtualized database servers in practice.

Query based SLA management

Current SLA management framework monitors replication delay of all queries on a virtualized database server as a whole. A much fine-grained approach would be monitoring replication delay of each query, optimizing process of each query, and guaranteeing SLAs on a query basis. In such cases, customers are able to specify SLA for each query, so that end-users could experience higher SLAs for some critical queries.

Bibliography

Abadi, D. (2009), ‘Data management in the cloud: Limitations and opportunities’, *Data Eng. Bull.* **32**(1), 3–12.

Abadi, D. (2012), ‘Consistency tradeoffs in modern distributed database system design: CAP is only part of the story’, *Computer* **45**(2), 37–42.

Agrawal, D., Abbadi, A. E., Emekci, F. and Metwally, A. (2009), Database management as a service: Challenges and opportunities, in *Proceedings of the 25th IEEE International Conference on Data Engineering*, ICDE ’09, IEEE Computer Society, Shanghai, China, pp. 1709–1716.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I. and Zaharia, M. (2010), ‘A view of cloud computing’, *Commun. ACM* **53**(4), 50–58.

Baker, J., Bond, C., Corbett, J. C., Furman, J., Khorlin, A., Larson, J., Leon, J.-M., Li, Y., Lloyd, A. and Yushprakh, V. (2011), Megastore: Providing scalable, highly available storage for interactive services, in *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, CIDR ’11, Asilomar, California, USA, pp. 223–234.

Bermbach, D., Klems, M., Tai, S. and Menzel, M. (2011), MetaStorage: A federated cloud storage system to manage consistency-latency tradeoffs, in *Proceedings of the*

2011 IEEE 4th International Conference on Cloud Computing, IEEE CLOUD '11, IEEE Computer Society, Washington, DC, USA, pp. 452–459.

Bermbach, D. and Tai, S. (2011), Eventual consistency: How soon is eventual? an evaluation of Amazon S3's consistency behavior, in *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, MW4SOC '11, ACM, Lisboa, Portugal, pp. 1:1–1:6.

Bernstein, P. A., Cseri, I., Dani, N., Ellis, N., Kalhan, A., Kakivaya, G., Lomet, D. B., Manne, R., Novik, L. and Talius, T. (2011), Adapting Microsoft SQL server for cloud computing, in *Proceedings of the 27th IEEE International Conference on Data Engineering*, ICDE '11, IEEE Computer Society, Hannover, Germany, pp. 1255–1263.

Bodik, P., Fox, A., Franklin, M. J., Jordan, M. I. and Patterson, D. A. (2010), Characterizing, modeling, and generating workload spikes for stateful services, in *Proceedings of the 1st ACM Symposium on Cloud computing*, SoCC '10, ACM, Indianapolis, IN, USA, pp. 241–252.

Brantner, M., Florescu, D., Graf, D., Kossmann, D. and Kraska, T. (2008), Building a database on S3, in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, ACM, Vancouver, BC, Canada, pp. 251–264.

Brewer, E. (2000), Towards robust distributed systems (abstract), in *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, ACM, Portland, OR, USA, p. 7.

Brewer, E. (2012), 'CAP twelve years later: How the "rules" have changed', *Computer* **45**(2), 23–29.

Broberg, J., Buyya, R. and Tari, Z. (2009), 'MetaCDN: Harnessing 'storage clouds' for high performance content delivery', *J. Netw. Comput. Appl.* **32**(5), 1012–1022.

Burrows, M. (2006), The Chubby lock service for loosely-coupled distributed systems, in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, USENIX Association, Seattle, WA, USA, pp. 335–350.

Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J. and Brandic, I. (2009), 'Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility', *Future Gener. Comput. Syst.* **25**(6), 599–616.

Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., Simitci, H., Haridas, J., Uddaraju, C., Khatri, H., Edwards, A., Bedekar, V., Mainali, S., Abbasi, R., Agarwal, A., Haq, M. F. u., Haq, M. I. u., Bhardwaj, D., Dayanand, S., Adusumilli, A., McNett, M., Sankaran, S., Manivannan, K. and Rigas, L. (2011), Windows Azure Storage: a highly available cloud storage service with strong consistency, in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, ACM, Cascais, Portugal, pp. 143–157.

Cattell, R. (2011), 'Scalable SQL and NoSQL data stores', *SIGMOD Rec.* **39**(4), 12–27.

Cecchet, E., Singh, R., Sharma, U. and Shenoy, P. (2011), Dolly: virtualization-driven database provisioning for the cloud, in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, ACM, Newport Beach, CA, USA, pp. 51–62.

Chandra, T. D., Griesemer, R. and Redstone, J. (2007), Paxos made live: an engineering perspective, in *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, ACM, Portland, Oregon, USA, pp. 398–407.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R. E. (2008), 'Bigtable: A distributed storage system for structured data', *ACM Trans. Comput. Syst.* **26**(2), 4:1–4:26.

Chohan, N., Bunch, C., Pang, S., Krintz, C., Mostafa, N., Soman, S. and Wolski, R. (2009), AppScale: Scalable and open AppEngine application development and deployment, in D. R. Avresky, M. Diaz, A. Bode, B. Ciciani and E. Dekel, eds, *Proceedings of the 1st International Conference on Cloud Computing*, Vol. 34 of *CloudComp '09*, Springer Berlin Heidelberg, Munich, Germany, pp. 57–70.

Cipar, J., Ganger, G., Keeton, K., Morrey, III, C. B., Soules, C. A. and Veitch, A. (2012), LazyBase: trading freshness for performance in a scalable database, in *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, ACM, Bern, Switzerland, pp. 169–182.

Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I. and Warfield, A. (2005), Live migration of virtual machines, in *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation*, Vol. 2 of *NSDI '05*, USENIX Association, Berkeley, CA, USA, pp. 273–286.

Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D. and Yerneni, R. (2008), 'PNUTS: Yahoo!'s hosted data serving platform', *Proc. VLDB Endow.* **1**(2), 1277–1288.

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R. and Sears, R. (2010), Benchmarking cloud serving systems with YCSB, in *Proceedings of the 1st ACM Symposium on Cloud computing*, SoCC '10, ACM, Indianapolis, IN, USA, pp. 143–154.

Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R. and Woodford, D. (2012), Spanner: Google's globally-distributed database, in *Proceedings of the 10th USENIX conference on Op-*

erating Systems Design and Implementation, OSDI '12, USENIX Association, Berkeley, CA, USA, pp. 251–264.

Cunha, I., Almeida, J., Almeida, V. and Santos, M. (2007), Self-adaptive capacity management for multi-tier virtualized environments, in *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management*, IM '07, IEEE Computer Society, Munich, Germany, pp. 129–138.

Das, S., Agrawal, D. and El Abbadi, A. (2009), ElasTraS: an elastic transactional data store in the cloud, in *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud '09, USENIX Association, Berkeley, CA, USA, pp. 1–5.

Das, S., Agrawal, D. and El Abbadi, A. (2010), G-Store: a scalable data store for transactional multi key access in the cloud, in *Proceedings of the 1st ACM Symposium on Cloud computing*, SoCC '10, ACM, New York, NY, USA, pp. 163–174.

Das, S., Nishimura, S., Agrawal, D. and El Abbadi, A. (2011), 'Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration', *Proc. VLDB Endow.* **4**(8), 494–505.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W. (2007), 'Dynamo: Amazon's highly available key-value store', *SIGOPS Oper. Syst. Rev.* **41**(6), 205–220.

Durkee, D. (2010), 'Why cloud computing will never be free', *Commun. ACM* **53**(5), 62–69.

Elmore, A. J., Das, S., Agrawal, D. and El Abbadi, A. (2011), Zephyr: live migration in shared nothing databases for elastic cloud platforms, in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, ACM, Athens, Greece, pp. 301–312.

Evangelinos, C. and Hill, C. N. (2008), Cloud computing for parallel scientific HPC applications: Feasibility of running coupled atmosphere-ocean climate models on Amazon's EC2, in *Proceedings of the 1st Workshop on Cloud Computing and Its Applications*, CCA '08, Chicago, IL, USA.

Floratou, A., Patel, J. M., Lang, W. and Halverson, A. (2011), When free is not really free: what does it cost to run a database workload in the cloud?, in *Proceedings of the 3rd TPC Technology Conference on Topics in Performance Evaluation, Measurement and Characterization*, TPCTC '11, Springer, Seattle, WA, USA, pp. 163–179.

Florescu, D. and Kossmann, D. (2009), 'Rethinking cost and performance of database systems', *SIGMOD Rec.* **38**(1), 43–48.

Ghemawat, S., Gobioff, H. and Leung, S.-T. (2003), 'The Google file system', *SIGOPS Oper. Syst. Rev.* **37**(5), 29–43.

Gilbert, S. and Lynch, N. (2002), 'Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services', *SIGACT News* **33**(2), 51–59.

Gilbert, S. and Lynch, N. (2012), 'Perspectives on the CAP theorem', *Computer* **45**(2), 30–36.

Gray, J. (2008), 'Distributed computing economics', *Queue* **6**(3), 63–68.

Guo, W., Sun, W., Jin, Y., Hu, W. and Qiao, C. (2010), 'Demonstration of joint resource scheduling in an optical network integrated computing environment', *Comm. Mag.* **48**(5), 76–83.

Habib, I. W., Song, Q., Li, Z. and Rao, N. S. (2006), 'Deployment of the GMPLS control plane for grid applications in experimental high-performance networks', *Comm. Mag.* **44**(3), 65–73.

Hacigümüs, H., Mehrotra, S. and Iyer, B. R. (2002), Providing database as a service, in *Proceedings of the 18th IEEE International Conference on Data Engineering*, ICDE '02, IEEE Computer Society, San Jose, CA, USA, pp. 29–38.

Hey, T., Tansley, S. and Tolle, K. M., eds (2009), *The Fourth Paradigm: Data-Intensive Scientific Discovery*, Microsoft Research, Redmond, Washington, USA.

Hill, Z. and Humphrey, M. (2009), A quantitative analysis of high performance computing with Amazon's EC2 infrastructure: The death of the local cluster?, in *Proceedings of the 10th IEEE/ACM International Conference on Grid Computing*, IEEE Computer Society, Banff, AB, Canada, pp. 26–33.

Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M. and Lewin, D. (1997), Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web, in *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, STOC '97, ACM, El Paso, TX, USA, pp. 654–663.

Keeton, K., Morrey, III, C. B., Soules, C. A. and Veitch, A. (2010), 'LazyBase: freshness vs. performance in information management', *SIGOPS Oper. Syst. Rev.* **44**(1), 15–19.

Kemme, B., Peris, R. J. and Patiño-Martínez, M. (2010), *Database Replication*, Synthesis Lectures on Data Management, 1st edn, Morgan & Claypool.

Kephart, J. O. and Chess, D. M. (2003), 'The vision of autonomic computing', *Computer* **36**(1), 41–50.

Kraska, T., Hentschel, M., Alonso, G. and Kossmann, D. (2009), 'Consistency rationing in the cloud: pay only when it matters', *Proc. VLDB Endow.* **2**(1), 253–264.

Krishnan, S. (2010), *Programming Windows Azure: Programming the Microsoft Cloud*, 1st edn, O'Reilly Media, Sebastopol, CA, USA.

Lakshman, A. and Malik, P. (2010), 'Cassandra: a decentralized structured storage system', *SIGOPS Oper. Syst. Rev.* **44**(2), 35–40.

Lehman, T., Sobieski, J. and Jabbari, B. (2006), 'DRAGON: a framework for service provisioning in heterogeneous grid networks', *Comm. Mag.* **44**(3), 84–90.

Lenk, A., Menzel, M., Lipsky, J., Tai, S. and Offermann, P. (2011), What are you paying for? performance benchmarking for Infrastructure-as-a-Service offerings, in *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*, IEEE CLOUD '11, IEEE Computer Society, Washington, DC, USA, pp. 484–491.

Levandoski, J. J., Lomet, D., Mokbel, M. F. and Zhao, K. K. (2011), Deuteronomy: Transaction support for cloud data, in *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, CIDR '11, Asilomar, California, USA, pp. 123–133.

Lloyd, W., Freedman, M. J., Kaminsky, M. and Andersen, D. G. (2011), Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS, in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, ACM, New York, NY, USA, pp. 401–416.

Lomet, D., Fekete, A., Weikum, G. and Zwilling, M. (2009), Unbundling transaction services in the cloud, in *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research*, CIDR '09, Asilomar, California, USA, pp. 1–10.

Lomet, D. and Mokbel, M. F. (2009), 'Locking key ranges with unbundled transaction services', *Proc. VLDB Endow.* **2**(1), 265–276.

Mell, P. M. and Grance, T. (2011), Sp 800-145. the NIST definition of cloud computing, Technical report, National Institute of Standards and Technology, Gaithersburg, MD, USA.

Padala, P., Shin, K. G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A. and Salem, K. (2007), Adaptive control of virtualized resources in utility computing environments, in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '07, ACM, Lisboa, Portugal, pp. 289–302.

Ranjan, R. and Zhao, L. (2011), 'Peer-to-Peer service provisioning in cloud computing environments', *J. Supercomput.* **Online First**, 1–31.

Ranjan, R., Zhao, L., Wu, X., Liu, A., Quiroz, A. and Parashar, M. (2010), Peer-to-Peer cloud provisioning: Service discovery and load-balancing, in N. Antonopoulos and L. Gillam, eds, *Cloud Computing: Principles, Systems and Applications*, Vol. 0 of *Computer Communications and Networks*, Springer, pp. 195–217.

Ristenpart, T., Tromer, E., Shacham, H. and Savage, S. (2009), Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds, in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, ACM, Chicago, IL, USA, pp. 199–212.

Rogers, J., Papaemmanouil, O. and Cetintemel, U. (2010), A generic auto-provisioning framework for cloud databases, in *Proceedings of the 26th IEEE International Conference on Data Engineering Workshops*, ICDEW '10, IEEE Computer Society, Long Beach, CA, USA, pp. 63–68.

Sakr, S. and Liu, A. (2012), SLA-based and consumer-centric dynamic provisioning for cloud databases, in *Proceedings of the 5th IEEE International Conference on Cloud Computing*, IEEE CLOUD '12, IEEE Computer Society, Honolulu, HI, USA, pp. 360–367.

Sakr, S., Zhao, L., Wada, H. and Liu, A. (2011), CloudDB AutoAdmin: Towards a truly elastic cloud-based data store, in *Proceedings of the 9th IEEE International Confer-*

ence on Web Services, ICWS '11, IEEE Computer Society, Washington, DC, USA, pp. 732–733.

Schad, J., Dittrich, J. and Quiané-Ruiz, J.-A. (2010), 'Runtime measurements in the cloud: observing, analyzing, and reducing variance', *Proc. VLDB Endow.* **3**(1-2), 460–471.

Shute, J., Oancea, M., Ellner, S., Handy, B., Rollins, E., Samwel, B., Vingralek, R., Whipkey, C., Chen, X., Jegerlehner, B., Littlefield, K. and Tong, P. (2012), F1 - the fault-tolerant distributed RDBMS supporting Google's ad business, in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, ACM, New York, NY, USA, pp. 777–778.

Sobel, W., Subramanyam, S., Sucharitakul, A., Nguyen, J., Wong, H., Patil, S., Fox, A. and Patterson, D. (2008), Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0, in *Proceedings of the 1st Workshop on Cloud Computing and Its Applications*, CCA '08, Chicago, IL, USA.

Soror, A. A., Minhas, U. F., Aboulnaga, A., Salem, K., Kokosielis, P. and Kamath, S. (2008), 'Automatic virtual machine configuration for database workloads', *ACM Trans. Database Syst.* **35**(1), 7:1–7:47.

Sovran, Y., Power, R., Aguilera, M. K. and Li, J. (2011), Transactional storage for geo-replicated systems, in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, ACM, New York, NY, USA, pp. 385–400.

Stonebraker, M., Bear, C., Çetintemel, U., Cherniack, M., Ge, T., Hachem, N., Harizopoulos, S., Lifter, J., Rogers, J., and Zdonik, S. (2007), One size fits all? – part 2: Benchmarking results, in *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research*, CIDR '07, Asilomar, California, USA, pp. 173–184.

Stonebraker, M. and Cetintemel, U. (2005), ”one size fits all”: An idea whose time has come and gone, in *Proceedings of the 21st IEEE International Conference on Data Engineering*, ICDE ’05, IEEE Computer Society, Washington, DC, USA, pp. 2–11.

Suleiman, B., Sakr, S., Jeffery, R. and Liu, A. (2012), ‘On understanding the economics and elasticity challenges of deploying business applications on public cloud infrastructure’, *J. Internet Serv. Appl.* **3**(2), 173–193.

Tanenbaum, A. S. and Steen, M. v. (2006), *Distributed Systems: Principles and Paradigms*, 2nd edn, Prentice Hall, Upper Saddle River, NJ, USA.

Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P. and Abadi, D. J. (2012), Calvin: fast distributed transactions for partitioned database systems, in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’12, ACM, New York, NY, USA, pp. 1–12.

Vaquero, L. M., Rodero-Merino, L., Caceres, J. and Lindner, M. (2008), ‘A break in the clouds: towards a cloud definition’, *SIGCOMM Comput. Commun. Rev.* **39**(1), 50–55.

Vogels, W. (2009), ‘Eventually consistent’, *Commun. ACM* **52**(1), 40–44.

Vouk, M. (2008), Cloud computing - issues, research and implementations, in *Proceedings of the 30th International Conference on Information Technology Interfaces*, ITI ’08, Dubrovnik, Croatia, pp. 31–40.

Weissman, C. D. and Bobrowski, S. (2009), The design of the force.com multitenant internet application development platform, in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’09, ACM, New York, NY, USA, pp. 889–896.

Wood, T., Shenoy, P., Venkataramani, A. and Yousif, M. (2007), Black-box and gray-box strategies for virtual machine migration, in *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI '07, USENIX Association, Cambridge, MA, USA, pp. 229–242.

Zhang, Q., Cheng, L. and Boutaba, R. (2010), 'Cloud computing: state-of-the-art and research challenges', *J. Internet Serv. Appl.* **1**(1), 7–18.