A Framework for Consumer-Centric SLA Management of Cloud-Hosted Databases

Liang Zhao *†, Sherif Sakr *†, Anna Liu *†
*National ICT Australia (NICTA)
†School of Computer Science and Engineering
University of New South Wales, Australia
{firstname.lastname}@Nicta.com.au

***Abstract*— 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, we are witnessing a proliferation in the number of cloud-hosted applications with a tremendous increase in the scale of the data generated as well as being consumed by such applications. Cloud-hosted database systems powering these applications form a critical component in the software stack of these applications. *Service Level Agreements* (SLA) represent the contract which captures the agreed upon guarantees between a service provider and its customers. The specifications of existing service level agreements (SLA) for cloud services are not designed to flexibly handle even relatively straightforward performance and technical requirements of consumer applications. In this article, we present a novel approach for SLA-based management of cloud-hosted databases from the *consumer* perspective. We present an end-to-end framework for *consumer-centric* SLA management of cloud-hosted databases. 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. In this framework, the SLA of the consumer 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 (scaling out/in the database tier) when required. The framework is database platform-agnostic, uses virtualization-based database replication mechanisms and requires zero source code changes of the cloud-hosted software applications. The experimental results demonstrate the effectiveness of our SLA-based framework in providing the consumer applications with the required flexibility for achieving their SLA requirements.**

***Index Terms*— Service Level Agreements (SLA), Cloud Databases, NoSQL Systems, Database-as-a-Service.**

I. Introduction

Cloud computing technology represents a new paradigm for the provisioning of computing infrastructure. This paradigm shifts the location of this infrastructure 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 [1] where the economy of scale principles help to effectively drive down the cost of computing infrastructure. 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 (virtually) unlimited resources and infinite scalability. Hence, it becomes possible, at least theoretically, to achieve unlimited throughput by continuously adding computing resources (e.g. database servers) if the workload increases.

In practice, the advantages of the cloud computing paradigm opens up new avenues for deploying novel applications which were not economically feasible in a traditional enterprise infrastructure setting. Therefore, the cloud has become an increasingly popular platform for hosting software applications in a variety of domains such as e-retail, finance, news and social networking. Thus, we are witnessing a proliferation in the number of applications with a tremendous increase in the scale of the data generated as well as being consumed by such applications. *Cloud-hosted database systems powering these applications form a critical component in the software stack of these applications*.

Cloud computing is by its nature a fast changing environment which is designed to provide services to unpredictably diverse sets of clients and heterogenous workloads. Several studies have also reported that the variation of the performance of cloud computing resources is high [2], [3]. These characteristics raise serious concerns from the cloud consumers' perspective regarding the manner in which the SLA of their application can be managed. According to a Gartner market report released in November 2010, SaaS is forecast to have a 15.8% growth rate through 2014 which makes SaaS and cloud very interesting to the services industry, but the viability of the business models depends on the practicality and the success of the terms and conditions (SLAs) being offered by the service provider(s) in addition to their satisfaction to the service consumers. Therefore, successful SLA management is a critical factor to be considered by both providers and consumers alike. Existing service level agreements (SLAs) of cloud providers are not designed for supporting the straightforward requirements and restrictions under which SLA of consumers' applications need to be handled. Particularly, most providers guarantee only the availability (but not the performance) of their services [4]. Therefore, consumer concerns on SLA handling for their cloud-hosted databases along with the limitations of existing SLA frameworks to express and enforce SLA requirements in an automated manner creates the need for SLA-based management techniques for cloud-hosted

databases. In this article, we present a novel approach for SLA-based management of cloud-hosted databases from the *consumer* perspective. In particular, we summarize the main contributions of this article as follows:

- We present an overview of the different options of hosting the database tier of software applications in cloud environments. In addition, we provide a detailed discussion for the main challenges for managing and achieving the SLA requirements of cloud-hosted database services.
- We present the design and implementation details of an end-to-end framework that enables the cloud consumer applications to *declaratively* define and manage their SLA for the cloud-hosted database tiers in terms of goals which are subjected to a number of constraints that are specific to their application requirements. The presented framework is database platform-agnostic and relies on virtualization-based database replication mechanism.
- We present consumer-centric dynamic provisioning mechanisms for cloud-hosted databases based on adaptive application requirements for two significant SLA metrics, namely, data freshness and transactions response times.
- We conduct an extensive set of experiments that demonstrate the effectiveness of our framework in providing the cloud consumer applications with the required flexibility for achieving their SLA requirements of the cloud-hosted databases.

To the best of our knowledge, our approach is the first that tackles the problem of SLA management for cloud-hosted databases from the perspective of the *consumer* applications. The remainder of this article is structured as follows. Section II provides an overview of the different options of deploying the database tier of software applications on cloud platforms. Section III discusses the main challenges of SLA Management for cloud-hosted databases. Section IV presents an overview of our framework architecture for consumer-centric SLA management. The main challenges of database replication management in cloud environments are discussed in Section V. An experimental evaluation for the performance characteristics of database replication in virtualized cloud environments is presented in Section VI. Our mechanism for provisioning the database tier based on the consumer-centric SLA metric of data freshness is presented in Section VII and for the SLA metric of the response times of application transactions is presented in Section VIII. Section IX summarizes the related work before we conclude the article in Section X.

II. OPTIONS OF HOSTING CLOUD DATABASES

Over the past decade, rapidly growing Internet-based services such as e-mail, blogging, social networking, search and e-commerce have substantially redefined the way consumers communicate, access contents, share information and purchase products. In principle, the main goal of cloud-based data management systems is to facilitate the job of implementing every application as a distributed, scalable and widely-accessible service on the Web. In practice, there are three different approaches for hosting the database tier of software applications in cloud platforms, namely, the platform storage services (NoSQL systems), the relational database as a service (DaaS) and virtualized database servers. In the following subsections, we discuss the capabilities and limitations of each of these approaches.

A. The platform storage services

This approach relies on a new wave of storage platforms named as key-value stores or NoSQL (Not Only SQL) systems. These systems are designed to achieve high throughput and high availability by giving up some functionalities that traditional database systems offer such as joins and ACID transactions [5]. For example, most of the NoSQL systems provide simple call level data access interfaces (in contrast to a SQL binding) and rely on weaker consistency management protocols (e.g. eventual consistency [6]). Commercial cloud offerings of this approach include Amazon SimpleDB and Microsoft Azure Table Storage. In addition, there is a large number of open source projects that have been introduced which follow the same principles of NoSQL systems [7] such as HBase and Cassandra. In practice, migrating existing software application that uses relational database to NoSQL offerings would require substantial changes in the software code due to the differences in the data model, query interface and transaction management support. In addition, developing applications on top of an eventually consistent NoSQL datastore requires a higher effort compared to traditional databases because they hinder important factors such as data independence, reliable transactions, and other cornerstone characteristics often required by applications that are fundamental to the database industry [8], [9]. In practice, the majority of today's platform storage systems are more more suitable for OLAP applications than for OLTP applications [10].

B. Relational Database as a Service (DaaS)

In this approach, a third party service provider hosts a relational database as a service [11]. Such services alleviate the need for their users to purchase expensive hardware and software, deal with software upgrades and hire professionals for administrative and maintenance tasks. Cloud offerings of this approach include Amazon RDS and Microsoft SQL Azure. For example, Amazon RDS provides access to the capabilities of MySQL or Oracle database while Microsoft SQL Azure has been built on Microsoft SQL Server technologies. As such, users 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 RDBMS of the existing software application is compatible with the offered service. However, many relational database systems are, as yet, not supported by the DaaS paradigm (e.g. DB2, Postgres). In addition, some limitations or restrictions might be introduced by the service provider for different reasons[1]. Moreover, the *consumer* applications do not have sufficient flexibility in controlling the allocated

[1] http://msdn.microsoft.com/en-us/library/windowsazure/ee336245.aspx

resources of their applications (e.g. dynamically allocating more resources for dealing with increasing workload or dynamically reducing the allocated resources in order to reduce the operational cost). The whole resource management and allocation process is controlled at the provider side which limits the ability of the consumer applications to maximize their benefits from the elasticity and scalability features of the cloud environment.

C. Virtualized Database Server

Virtualization is a key technology of the cloud computing paradigm. Virtual machine technologies are increasingly being used to improve the manageability of software systems and lower their total cost of ownership. They allow resources to be allocated to different applications on demand and hide the complexity of resource sharing from cloud users by providing a powerful abstraction for application and resource provisioning. In particular, resource virtualization technologies add a flexible and programmable layer of software between applications and the resources used by these applications. The *virtualized database server* approach takes an existing application that has been designed to be used in a conventional data center, and then port 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 approach, database servers, like any other software components, are migrated to run in virtual machines. *Our work presented in this article belongs to this approach*. In principle, one of the major advantages of the *virtualized database server* approach is that the application can have full control in dynamically allocating and configuring the physical resources of the database tier (database servers) as needed [12], [13], [14]. Hence, software applications can fully utilize the elasticity feature of the cloud environment to achieve their defined and customized scalability or cost reduction goals. However, achieving these goals requires the existence of an admission control component which is responsible for monitoring the system state and taking the corresponding actions (e.g. allocating more/less computing resources) according to the defined application requirements and strategies. Several approaches have been proposed for building admission control components which are based on the *efficiency of utilization* of the allocated resources [13], [14]. In our approach, we focus on building an *SLA-based* admission control component as a more practical and *consumer-centric* view for achieving the requirements of their applications.

III. Challenges of SLA Management for Cloud-Hosted Databases

An SLA is a contract between a service provider and its customers. *Service Level Agreements* (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) (e.g. maximum response times, minimum throughput rates, 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 article, these parties are defined as follows:

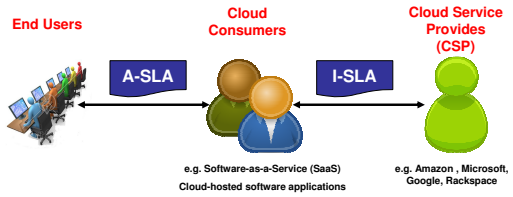


Fig. 1: SLA Parties in Cloud Environments

- *Cloud Service Providers (CSP)*: They offer client-provisioned and metered computing resources (e.g. CPU, storage, memory, network) that can be rented for flexible time durations. In particular, they include: Infrastructure-as-a-Service providers (IaaS), Platform-as-a-Service providers (PaaS) and Database-as-a-Service (DaaS). Examples are: Amazon, Microsoft and Google.
- *Cloud Consumers*: They represent the cloud-hosted software applications that utilize the services of CSP and are financially responsible for their resource consumptions.
- *End Users*: They represent the legitimate users for the services (applications) that are offered by cloud consumers.

While cloud service providers charge cloud consumers for renting computing resources to deploy their applications, cloud consumers may charge their end users for processing their workloads (e.g. SaaS) or may process the user requests for free (cloud-hosted business application). In both cases, the cloud consumers need to guarantee their users' SLA. Penalties are applied in the case of SaaS and reputation loss is incurred in the case of cloud-hosted business applications. For example, Amazon found that every 100ms of latency costs them 1% in sales and Google found that an extra 500ms in search page generation time dropped traffic by 20%[2]. In addition, large enterprise web applications (e.g., eBay and Facebook) need to provide high assurances in terms of SLA metrics such as response times and service availability to their users. Without such assurances, service providers of these applications stand to lose their user base, and hence their revenues.

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

1) *Cloud Infrastructure SLA (I-SLA)*: These SLA are offered by cloud providers to cloud consumers to assure the quality levels of their cloud computing resources (e.g.,

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

server performance, network speed, resources availability, storage capacity).

2) *Cloud-hosted 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 consumers often offer such guarantees to their application's end users in order to assure the quality of services that are offered such as the application's response time and data freshness.

Figure 1 illustrates the relationship between *I-SLA* and *A-SLA* in the software stack of cloud-hosted applications. In practice, traditional cloud monitoring technologies (e.g. Amazon CloudWatch focus on low-level computing resources (e.g. CPU speed, CPU utilization, disk speed). In principle, 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 consumer applications of cloud resources 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 [15]. Therefore, predicting the workload behavior (e.g. arrival pattern, I/O behavior, service time distribution) and consequently accurate planning of the computing resource requirements are very challenging tasks.
- *Performance variability of cloud resources*: Several studies have reported that the variation of the performance of cloud computing resources is high [2], [3]. As a result, currently, cloud service providers do not provide adequate SLAs for their service offerings. Particularly, most providers guarantee only the availability (but not the performance) of their services [1], [16].
- *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 [17].

In practice, it is a very challenging goal to delegate the management of the SLA requirements of the consumer applications to the side of 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 consumer applications (tenants) that can be simultaneously running in a cloud environment. Therefore, it becomes a significant issue for the cloud consumers to be able to monitor and adjust the deployment of their systems if they intend to offer viable service level agreements (SLAs) to their customers (end users) [12]. Failing to achieve these goals will jeopardize the sustainable growth of cloud computing in the future and may result in valuable applications move away from the cloud. In the following sections, we present our *consumer-centric* approach for managing the SLA requirements of cloud-hosted databases.

IV. Framework Architecture

In this section, we present an overview of our consumer-centric framework that enables the cloud consumer applications to *declaratively* define and manage their SLA for the cloud-hosted database tiers in terms of goals which are subjected to a number of constraints that are specific to their application requirements. The framework also enables the consumer applications to declaratively define a set of application-specific rules (*action rules*) where the admission control component of the database tier needs to take corresponding actions in order to meet the expected system performance or to reduce the cost of the allocated cloud resources when they are not efficiently utilized. The framework continuously monitors the database workload, tracks the satisfaction of the application-defined SLA, evaluates the condition of the action rules and takes the necessary actions when required. The design principles of our framework architecture are to be *application-independent* and to require *no code modification* on the consumer software applications that the framework will support. In addition, the framework is database *platform-agnostic* and relies on *virtualization-based database replication* mechanism. In order to achieve these goals, we rely on a database proxying mechanism which provides the ability to forward database requests to the underlying databases using an intermediate piece of software, the proxy, and to return the results from those request transparently to the client program without the need of having any database drivers installed. 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.

In general, there exist many forms of SLAs with different metrics. In this article, we focus on the following two main consumer-centric SLA metrics:

- *Data freshness*: which represents the tolerated window of data staleness for each database replica. In other words, it represents the time between a committed update operation on the master database and the time when the operation is propagated and committed to the database replica (Section VII).
- *Transaction response time*: which represents the time between a transaction is presented to the database system and the time when the transaction execution is completed (Section VIII).

Figure 2 shows an overview of our framework architecture which consists of three main modules: the *monitor module*, the

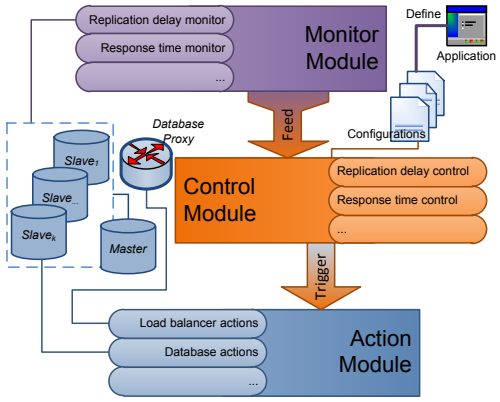


Fig. 2: Framework Architecture

control module and the *action module*. In this architecture, the consumer application is only responsible of configuring the control module of the framework by declaratively defining (using an XML dialect) the specifications of the SLA metrics of their application. In addition, the consumer application declaratively defines (using another XML dialect) a set of rules that specify the actions that should be taken (when a set of conditions are satisfied) in order to meet the expected system performance or to reduce the cost of the allocated cloud resources when they are not efficiently utilized. More details and examples about the declarative definitions of the application-specific SLA metrics and action rules will be presented in Section VII and Section VIII. The control module also maintains the information about the configurations of the load balancer (e.g. proxy address, proxy script), the access information of each database replica (e.g. host address, port number) and the location information of each database replica (e.g. us-east, us-west, eu-west). On the runtime, the monitor module is responsible of continuously tracking the application-defined SLAs and feeding the control module with the collected information. The control module is responsible for continuously checking the monitored SLA values against their associated application-defined SLAs and triggers the action module to scale out/in the database according to the application-defined action rules.

In general, dynamic provisioning at the database-tier involves increasing or decreasing the number of database servers allocated to an application in response to workload changes. *Data replication* is a well-known strategy to achieve the availability, scalability and performance improvement goals in the data management world. In particular, when the application load increases and the database tier becomes the *bottleneck* in the stack of the software application, there are two main options for achieving scalability at the database tier to enable the application to cope with more client requests:

1) *Scaling up (Vertical scalability)*: which aims at allocating a bigger machine with more horsepower (e.g. more processors, memory, bandwidth) to act as a database server.
2) *Scaling out (Horizontal scalability)*: which aims at replicating the data across more machines.

In practice, 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, it is both extensible and economical, especially in a dynamic workload environment, to scale out by adding another commodity server which fits well with the pay-as-you-go pricing philosophy of cloud computing. In addition, the scale out mechanism is more adequate for achieving the elasticity benefit of cloud platforms by facilitating the process of horizontally adding or removing (in case of scaling in), as necessary, computing resources according to the application workload and requirements.

In database replication, there are two main replication strategies: *master-slave* and *multi-master*. In master-slave, updates are sent to a single master node and lazily replicated to slave nodes. Data on slave nodes might be stale and it is the responsibility of the application to check for data freshness when accessing a slave node. Multi-master replication enforces a serializable execution order of transactions between all replicas so that each of them applies update transactions in the same order. This way, any replica can serve any read or write request. Our framework mainly considers the master-slave architecture as it is the most common architecture employed by most web applications in the cloud environment. For the sake of simplicity of achieving the consistency goal among the database replicas and reducing the effect of network communication latency, we employ the ROWA (read-once-write-all) protocol on the Master copy [18]. However, our framework can be easily extended to support the multi-master replication strategy as well.

In general, provisioning of a new database replica involves extracting database content from an existing replica and copying that content to a new replica. In practice, the time taken to execute these operations mainly depends on the database size. To provision database replicas 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. Clearly, there is a tradeoff between the time to snapshot the database, the size of the transactional log and the amount of update transactions in the workload. In our framework this trade-off can be controlled by application-defined parameters. This tradeoff can be further optimized by applying recently proposed live database migration techniques [19], [20].

V. DATABASE REPLICATION IN THE CLOUD

The **CAP** theorem [21] shows that a shared-data system can only choose at most two out of three properties: ***C***onsistency (all records are the same in all replicas), ***A***vailability (all replicas can accept updates or inserts), and tolerance to ***P***artitions (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 and accept update requests of data and at the same time cannot block the updates even while they read the same data for scalability reasons. Therefore, when data is replicated over a wide area, this essentially leaves just consistency and availability for a system to choose between. Thus, the **C** (consistency) part of **CAP** is typically compromised to yield reasonable system availability [10]. Hence, most of the cloud data management

overcome the difficulties of distributed replication by relaxing the consistency guarantees of the system. In particular, they implement various forms of weaker consistency models (e.g. eventual consistency [6]) so that all replicas do not have to agree on the same value of a data item at every moment of time. In particular, 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.

Florescu and Kossmann [22] have argued that in the new large scale web applications, the requirement to provide 100 percent read and write availability for all users has overshadowed the importance of the ACID paradigm as the gold standard for data consistency. In these applications, no user is ever allowed to be blocked. Hence, while having strong consistency mechanisms has been considered as a hard and expensive constraint in traditional database management systems, it has been turned into an optimization goal (that can be relaxed) in cloud-based database systems.

Kraska et al. [23] have presented a mechanism that allows software designers to define the consistency guarantees on the data instead of at the transaction level. In addition, it allows 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 (i.e., the penalty cost is low) and raise them when it matters (i.e., the penalty costs would be too high). Keeton et al. [24] have proposed a similar approach in a system called *LazyBase* that allows users to trade off query performance and result freshness. LazyBase breaks up metadata processing into a pipeline of ingestion, transformation and query stages which can be parallelized to improve performance and efficiency. LazyBase uses models of transformation and query performance to determine how to schedule transformation operations to meet users' freshness and performance goals, and to utilize resources efficiently.

While *Consistency Rationing* and *LazyBase* represent two approaches for supporting adaptive consistency management for cloud-hosted databases, they are more focused on the perspective of cloud service provider. On the contrary, our framework is more focused on the cloud consumer perspective. In particular, it provides the cloud consumer (software application) with flexible mechanisms for specifying and managing the extent of inconsistencies that they can tolerate. In addition, it allows the creation and monitoring of several replicas of the database with different levels of freshness across the different virtualized servers as we will show in a later section.

VI. Performance Evaluation of Database Replication on Virtualized Cloud Environments

In this section, we present an experimental evaluation for the performance characteristics of the master-slave database replication strategy on virtualized database server in cloud environments [25]. In particular, the main goals of the experiments of this section are:

- To investigate the scalability characteristics of the master-slave replication strategy with an increasing workload and an increasing number of database replicas in a virtualized cloud environment. In particular, we try to identify what factors act as limits on achievable scale in such deployments.
- To measure the average replication delay (window of data staleness) that could exist with an increasing number of database replicas and different configurations to the geographical locations of the slave databases.

A. Experiment design

The Cloudstone benchmark[3] 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 (e.g. browsing, searching and creating events), as well as, social operations (e.g. joining and tagging events)[26]. Unlike Web 1.0 applications, Web 2.0 applications behave differently on database in many ways. 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. Therefore, more write transactions are expected to be processed. Another difference is on the tolerance with regards to 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 (e.g. 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, we believe that the design and workload characteristics of the the Cloudstone benchmark is more suitable for the purpose of our study rather than other benchmarks such as TPC-W or RUBiS which are more representing 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 the load against the web application which in turn makes use of the database. The benchmark designs well for benchmarking performance of each tier for Web 2.0 applications. However, the original design of the benchmark makes it hard to push the database performance to its performance limits which limits its suitability for our experiments of focusing mainly on the database tier of the software stack. 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 pre-defined business logic before bypassing the request to the database tier. Thus the saturation on the web tier usually happens earlier than the saturation on the database tier. Therefore, we modified the design of the original software stack by removing the web server tier. In particular, we re-implemented the business logic of the application in a way that a user's operation can be processed directly at the database tier without any intermediate interpretation at the web server tier. Meanwhile, on top of our Cloudstone implementation, we also implemented a connection pool (i.e. DBCP[4]) and a proxy

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

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

(i.e. MySQL Connector/J[5]) components. 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 database replicas where all write operations are sent to the master while all read operations are distributed among slaves.

Multiple MySQL replicas are deployed to compose the database tier. For the purpose of monitoring replication delay in MySQL, we have created a *Heartbeats* database and a time/date function for each replica. The *Heartbeats* database, synchronized in the format of SQL statement across replicas, maintains a '*heartbeat*' table which records an *id* and a *timestamp* in each row. A heartbeat plug-in for Cloudstone is implemented to insert a new row with a global id and a local time stamp to the master periodically 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 time stamp. 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. We, therefore, implemented a user defined time/date function with a microsecond resolution that is based on a proposed solution to MySQL Bug #8523[6]. The clock synchronizations between the master and slaves are maintained by NTP[7] (Network Time Protocol) on Amazon EC2. We set the NTP protocol to synchronize with multiple time servers every second to have a better resolution.

With the customized Cloudstone[8] and the heartbeat plug-in, we are able to achieve our goal of measuring the end-to-end database throughput and the replication delay. In particular, we defined two configurations with read/write ratios of 50/50 and 80/20. We also defined three configurations of the geographical locations based on Availability Zones (they are distinct locations within a Region) and Regions (they are separated into geographic areas or countries) as follows: *same zone* where all slaves are deployed in the same Availability Zone of a Region of the master database; *different zones* where the slaves are in the same Region as the master database, but in different Availability Zones; *different regions* where all slaves are geographically distributed in a different Region from where the master database is located. The workload and the number of database replicas start with a small number and gradually increase at a fixed step. Both numbers stop increasing if there are no throughput gained.

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

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

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

[8]the source code of our Cloudstone customized implementation is available on http://code.google.com/p/clouddb-replication/

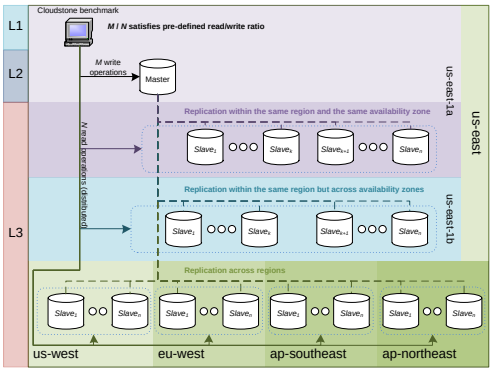


Fig. 3: Database replication on virtualized cloud servers.

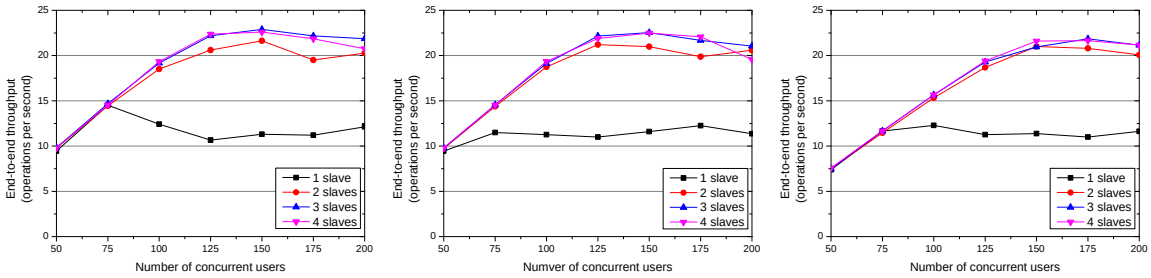
B. Experiment setup

We conducted our replication experiments in Amazon EC2 service with a three-layer implementation (Fig. 3). The first layer is the Cloudstone benchmark which 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 second layer includes the master database that 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 can be expected to be observed early. Both the master database server and the application benchmark are deployed in *us-east*-1*a* location. The third layer is a group of slaves which 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 achieved. Several options for the deployment locations of the slaves have been used, namely, the same zone as the master in *us-east*-1*a*, a different zone in *us-east*-1*b* 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 database replicas. Every run lasts 35 minutes, including 10-minute ramp-up, 20-minute steady stage and 5-minute ramp down. Moreover, for each run, both the master and slaves should start with a pre-loaded, fully-synchronized database.

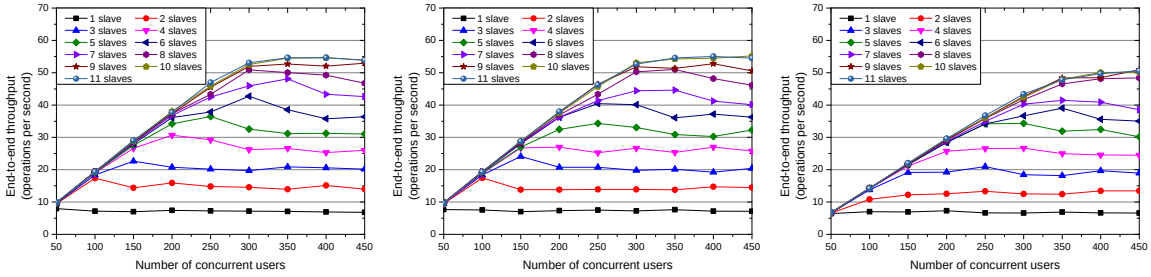
C. End-to-end throughput experiments

Fig. 4 and Fig. 5 show the throughput trends for up to 4 and 11 slaves with mixed configurations of three locations



(a) Same zone (*us-west-1a*) (b) Different zone (*us-west-1b*) (c) Different region (*eu-west-1a*)

Fig. 4: End-to-end throughput of the workload with the read/write ratio 50/50.



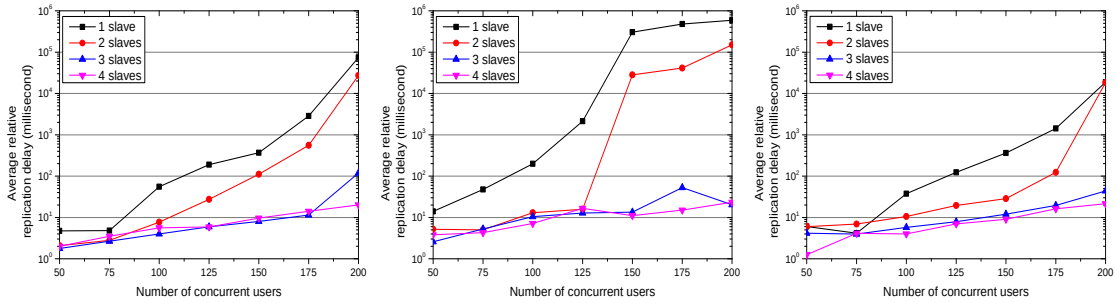
(a) Same zone (*us-west-1a*) (b) Different zone (*us-west-1b*) (c) Different region (*eu-west-1a*)

Fig. 5: End-to-end throughput of the workload with the read/write ratio 80/20.

and two read/write ratios. Both results indicate that MySQL with asynchronous master-slave replication is limited in its ability to scale due to the saturation to the master database. In particular, the throughput trends react to saturation movements and transitions in database replicas in regards to an increasing workload and an increasing number of database replicas. 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 an increasing number of slaves are synchronized to the master. Eventually, however, the saturation will transit from slaves to the master where the scalability limit is achieved. Taking throughput trends with configurations of the same zone and 50/50 ratio (Fig. 5a) as an example, the saturation point of 1 slave is initially observed at 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 maximum CPU utilization while the master's CPU usage rate is also approaching its utilization limit. Thus, ever since the 3rd slave is added, 175 workloads remain as the saturation point, but with the master being saturated instead of 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 keep up the read/write ratio that corresponds to the increment of the read throughput. Hence, the read throughput is suppressed by the benchmark, for the purpose of maintaining the pre-defined 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 the other two locations (Fig. 4b and Fig. 4c), and 10 slaves at 20/80 ratio in the same zone (Fig. 5a), and different zone (Fig. 5b) and also 9 slaves at 20/80 ratio in different regions (Fig. 5c).

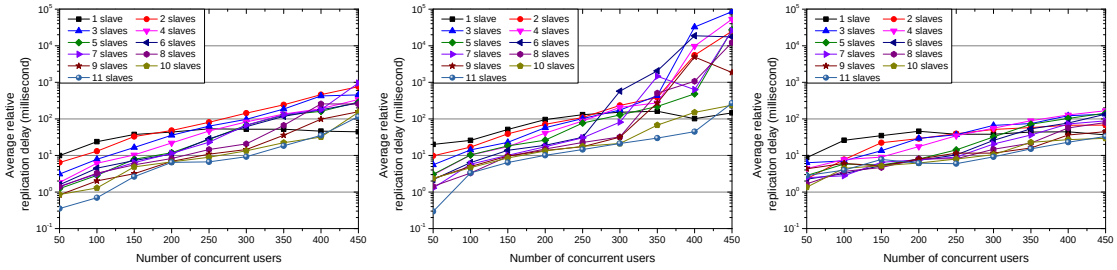
The configuration of the geographic locations is a factor that affects the end-to-end throughput, in the context of locations of users. In the case of our experiments, since all users emulated by Cloudstone send read operations from *us-east-1a*, distances between the users and the slaves increase, following the order of same zone, different zone and different region. Normally, a long distance incurs a slow round-trip time, which results in a smaller throughput for the same workload. Therefore, it can be expected that a decrease in maximum throughput would be observed when configurations of locations follow the order of same zone, different zone and different region. Moreover, the throughput degradation is also related to read percentages where higher read percentages would result in larger degradations. It explains why degradation of maximum throughput is more significant with configuration of 80/20 read/write ratio (Fig. 5). 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 databases in the cloud. For throughput trends of 1 slave at 50/50 read/write ratio with configurations of different zone and different region, respectively, if the configuration of locations is the only factor, the maximum throughput in the different zone (Fig. 4b) should be larger than the one in the different region (Fig. 4c). However, the main reason of throughput difference here is caused by the performance variation of instances rather than



(a) Same zone (*us-west-1a*) (b) Different zone (*us-west-1b*) (c) Different region (*eu-west-1a*)

Fig. 6: Average relative replication delay of the workload with the read/write ratio is 50/50.



(a) Same zone (*us-west-1a*) (b) Different zone (*us-west-1b*) (c) Different region (*eu-west-1a*)

Fig. 7: Average relative replication delay of the workload with the read/write ratio is 80/20.

the configuration of locations. The 1st slave from the same zone runs on top of a physical machine with an Intel Xeon CPU E5430 2.66GHz. While another 1st slave from the different zone is deployed in a physical machine powered by an Intel Xeon CPU E5507 2.27GHz. Because of the performance differences between physical CPUs, the slave from same zone performs better than the one from different zone. Previous research indicated that the coefficient of variation of CPU of small instances is 21% [2]. Therefore, it is a good strategy to validate instance performance before deploying applications into the cloud, as poor-performing instances are launched randomly and can largely affect application performance.

D. Replication delay experiments

Fig. 6 and Fig. 7 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 configurations of the geographical locations has a lower impact on the replication delay than that of the workload characteristics. The trends of the average relative replication delay respond to an increasing workload and an increasing number of database replicas. For most cases, with the number of database replicas being kept constant, the average relative replication delay surges along with an increasing workload which leads to more read and write operations sent to the slaves and the master database, respectively. It turns out that the increasing number of 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 results in higher replication delay. Similarly, the average relative replication delay decreases along with an increasing number of database replicas as the addition of a new slave leads to a reduction in the resource contention and subsequent decrease in replication delay.

As previously mentioned, the configuration of the geographic location of the slaves play a less significant role in affecting replication delay, in comparison to the changes 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 the *ping* command every second for a 20-minute period. The results suggest an average of 16, 21, and 173 milliseconds 1/2 round-trip time for the same zone (Fig. 6a and Fig. 7a), different zones (Fig. 6b and Fig. 7b) and different regions (Fig. 6c and Fig. 7c), respectively. However, The trends of the average relative replication delay can usually go up from two to four orders of magnitude (Fig. 6), or one to three orders of magnitude (Fig. 7). Therefore, it could be suggested that geographic replication would be applicable in the cloud as long as workload characteristics can be well managed (e.g. having a smart load balancer which is able to balance the operations based on estimated processing time).

VII. Provisioning The Database Tier Based on SLA of Data Freshness

A. Adaptive Replication Controller

In practice, the cost of maintaining several database replicas that are always strongly consistent is very high. Therefore,

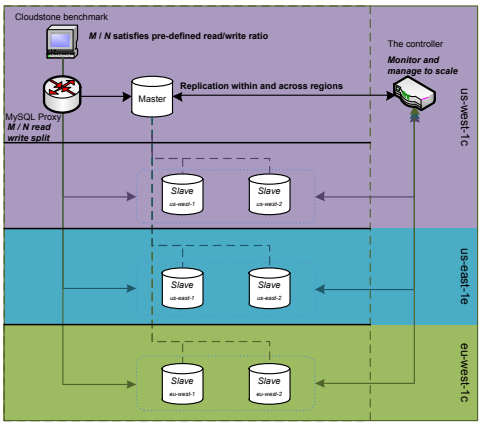


Fig. 8: Adaptive replication controller.

keeping several database replicas with different levels of freshness can be highly beneficial in the cloud environment since freshness can be exploited as an important metric of replica selection for serving the application requests as well as optimizing the overall system performance and monetary cost. Our framework provides the software applications with flexible mechanisms for specifying different service level agreements (SLA) of data freshness for the underlying database replicas. In particular, the framework allows specifying an SLA of data freshness for each database replica and continuously monitor the replication delay of each replica so that once a replica violates its defined SLA, the framework automatically activates another database replica at the closest geographic location in order to balance the workload and re-satisfy the defined SLA [27]. In particular, the SLA of the replication delay for each replica ($delay_{sla}$) is defined as an integer value in the unit of millisecond which represents two main components:

$$delay_{sla} = delay_{rtt} + delay_{tolerance}$$

where the round-trip time component of the SLA replication delay ($delay_{rtt}$) is the average round-trip time from the master to the database replica. 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 to be inconsistent. This tolerance component can vary from one replica to another depending on many factor such as the application requirements, the geographic location of the replica, and the workload characteristics and the load balancing strategy of each application. Therefore, the control module is responsible of triggering the action module for adding a new database replica, when necessary, in order to avoid any violation in the application-defined SLA of data freshness for the active database replicas. In our framework implementation, we follow an intuitive strategy that triggers the action module for adding a new replica when it detects a number of continuous up-to-date monitored replication delays of a replica which exceeds its application-defined threshold (T) of SLA violation of data freshness. In other words, for a running database replica, 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 (for the violating replica). It is worthy to note that the strategy of the control module in making the decisions (e.g. the timing, the placement, the physical creation) regarding the addition a new replica 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 article to investigate different strategies for making these decisions. We leave this aspect for future work.

B. Experimental Evaluation

We implemented two sets of experiments in order to evaluate the effectiveness of our adaptive replication controller in terms of its effect on the end-to-end system throughput and the replication delay for the underlying database replicas. Figure 8 illustrates the setup of our experiments in Amazon EC2 platform In the first set of experiments, we fix the value of the tolerance component ($delay_{tolerance}$) of the SLA replication delay to 1000 milliseconds and vary the monitor interval ($intvl_{mon}$) among the following set of values: 60, 120, 240 and 480 seconds. In the second set of experiments, we fix the monitor interval ($intvl_{mon}$) to 120 seconds and adjusts the SLA of replication delay ($delay_{sla}$) by varying the tolerance component of the replication delay ($delay_{tolerance}$) among the following set of values: 500, 1000, 2000 and 4000 milliseconds. We have been evaluating the round-trip component ($delay_{rtt}$) of the replication delays SLA ($delay_{sla}$) for the database replicas in the three geographical regions of our deployment by running *ping* command every second for a 10 minutes period. The resulting average three round-trip times ($delay_{rtt}$) are 30, 130 and 200 milliseconds for the master to slaves in *us-west*, *us-east* and *eu-west* respectively. Every experiment is executed for a period of 3000 seconds with a starting workload of 220 concurrent users and database requests with a read/write ratio of 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 replicas in 3 regions where each region hosts two replicas: the first replica is an active replica which is used from the start of the experiment for serving the database requests of the application while the second replica 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, we conducted two experiments without our adaptive replication controller in order to measure the end-to-end throughputs and replication delays of 3 (the minimum number of running replicas) and 6 (the maximum number of running replicas) slaves in order to measure the baselines of our comparison.

1) End-to-end throughput: Table I presents the end-to-end throughput results for our set of experiments with different configuration parameters. The *baseline* experiments represent

TABLE I: The effect of the the adaptive replication controller 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 replicas	Running time of all replicas (in seconds)	End-to-end throughput (operations per seconds)	Replication delay
Baselines with fixed number of replicas	*N/A*	*N/A*	3	9000	22.33	Fig. 9a
	N/A	*N/A*	6	18000	38.96	Fig. 9b
Varying the monitor interval ($intvl_{mon}$)	60	1000	$3 \rightarrow 6$	15837	38.43	Fig. 9d
	120	1000	$3 \rightarrow 6$	15498	36.45	Fig. 9c
	240	1000	$3 \rightarrow 6$	13935	34.12	Fig. 9e
	480	1000	$3 \rightarrow 6$	12294	31.40	Fig. 9f
Varying the tolerance of replication delay ($delay_{tolerance}$)	120	500	$3 \rightarrow 6$	15253	37.44	Fig. 9g
	120	1000	$3 \rightarrow 6$	15498	36.45	Fig. 9c
	120	2000	$3 \rightarrow 6$	13928	36.33	Fig. 9h
	120	4000	$3 \rightarrow 6$	14437	34.68	Fig. 9i

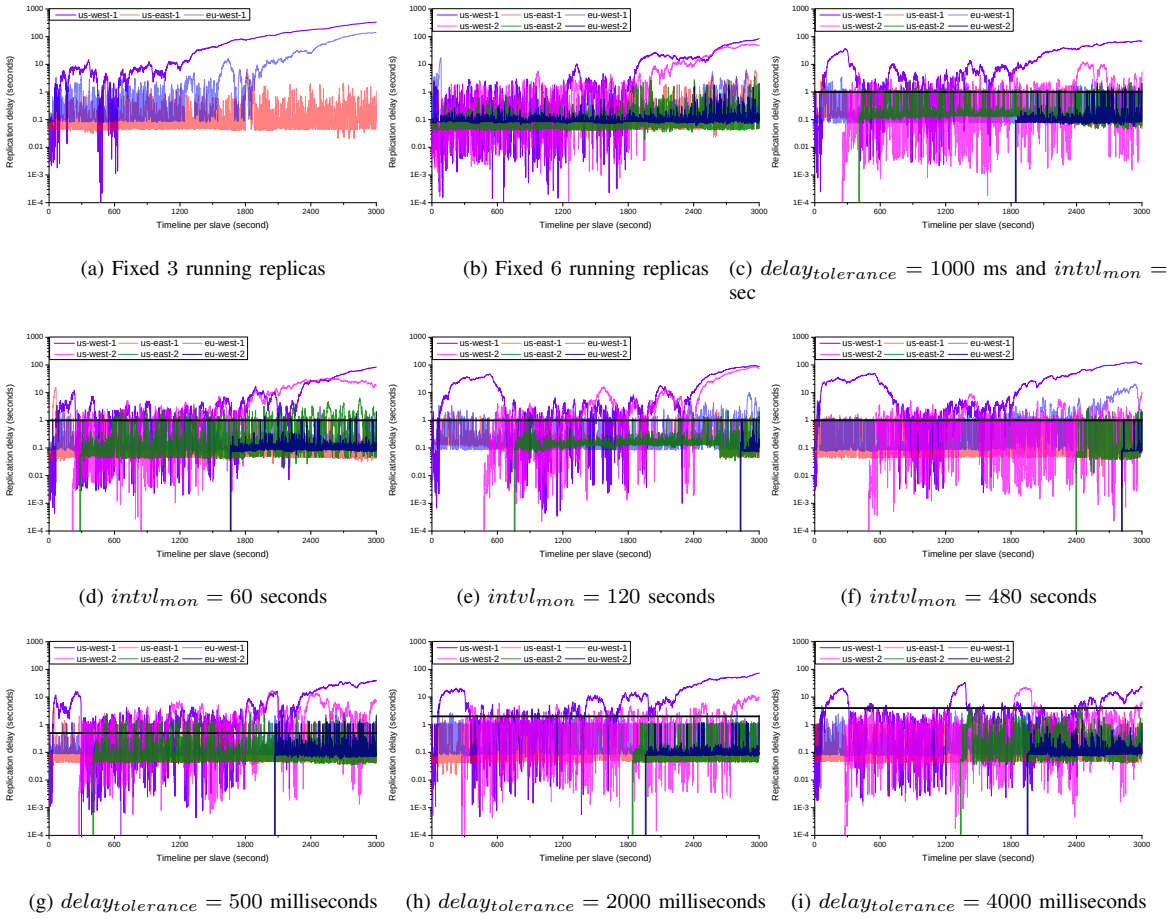
the *minimum* and *maximum* end-to-end throughput results with 22.33 and 38.96 operations per second, respectively. They also represent the minimum and maximum baseline for the running time of all database replicas with 9000 (3 running replicas, with 3000 seconds running time of each replica from the beginning to the end of the experiment) and 18000 (6 running replicas, with 3000 seconds running time of each replica) seconds, respectively. The end-to-end throughput of the other experiments fall between the two baselines based on the variance on the monitor interval ($intvl_{mon}$) and the tolerance of replication delay ($delay_{tolerance}$). Each experiment starts with 3 active replicas after which the number of replicas gradually increases during the experiments based on the configurations of the monitor interval and the SLA of replication delay parameters until it finally ends with 6 replicas. Therefore, the total running time of the database replicas for the different experiments fall within the range between 9000 and 18000 seconds. Similarly, the end-to-end throughput delivered by the adaptive replication controller for the different experiments fall within the end-to-end throughput range produced by the two baseline experiments of 22.33 and 38.96 operations per second. 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 database replicas. 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 replicas 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, having the values of the tolerance of the replication delay equal to 4000 and 1000 result in longer running times of the database replicas than having the values equal to 2000 and 500. On the other side, the increase of running time of all replicas shows a linear trend along with the increase of the end-to-end throughput. However, a general conclusion might not easy to draw because the trend is likely affected by the workload characteristics.

2) Replication delay: Figure 9 illustrates the effect of the adaptive replication controller on the performance of the replication delay for the cloud-hosted database replicas. Figure (9a) and Figure (9b) show the replication delay of the two baseline cases for our comparison. They represent the experiments of running with a fixed number of replicas (3 and 6 respectively) from the starting times of the experiments to their end times. Figure (9a) shows that the replication delay tends to follow different patterns for the different replicas. The two trends of *us-west-1* and *eu-west-1* surge significantly at 260 and 280 users, respectively. On the same time, the trend of *us-east-1* tends to be stable throughout the entire running time of the experiment. The main reason behind this is the performance variation between the hosting EC2 instances for the database replicas[9]. Due to the performance differences between the physical CPUs specifications, *us-east-1* is able to handle the amount of operations that saturate *us-west-1* and *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 instances. As illustrated in Figure (8), the MySQL Proxy location is closer to *us-west-1* than *eu-west-1*. Therefore, the forwarded database operations by the MySQL Proxy take less time to arrived at *us-west-1* than *eu-west-1* which leads to more congestion on the *us-west-1* side. Similarly, in Figure (9b), the replication delay tends to surge in both *us-west-1* and *us-west-2* for the same reason of the difference in the geographic location of the underlying database replica.

Figures (9c), and (9g) to (9i) show the results of the replication delay for our experiments using different values for the monitor interval ($intvl_{mon}$) and the tolerance of replication delay ($delay_{tolerance}$) parameters. For example, Figure (9c) shows that the *us-west-2*, *us-east-2*, and *eu-west-2* replicas are added in sequence at the 255^{th}, 407^{th} and 1843^{th} seconds, where the drop lines are emphasized. The addition of the three replicas are caused by the SLA-violation of the *us-west-1* replicas at different periods. In particular, there are four SLA-violation periods for *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: 1) 67:415 (total of 349 seconds). 2) 670:841 (total of 172 seconds). 3) 1373:1579 (total of 207 seconds). 4) 1615:3000 (total of 1386 seconds). The addition of new replicas is only triggered on the 1^{st} and the 4^{th} periods based on the time point analysis. The 2^{nd} and the 3^{rd} periods do not trigger the addition of new replica as the number of detected SLA violations does not exceed the defined threshold (T).

Figures (9c), and (9d) to (9f) show the effect of varying the monitor interval ($intvl_{mon}$) on the replication delay of the

[9]Both *us-west-1* and *eu-west-1* are powered by Intel(R) Xeon(R) CPU E5507 @ $2.27GHz$, whereas *us-east-1* is deployed with a better CPU, Intel(R) Xeon(R) CPU E5645 @ $2.40GHz$



(a) Fixed 3 running replicas (b) Fixed 6 running replicas (c) $delay_{tolerance} = 1000$ ms and $intvl_{mon} = 120$ sec

(d) $intvl_{mon} = 60$ seconds (e) $intvl_{mon} = 120$ seconds (f) $intvl_{mon} = 480$ seconds

(g) $delay_{tolerance} = 500$ milliseconds (h) $delay_{tolerance} = 2000$ milliseconds (i) $delay_{tolerance} = 4000$ milliseconds

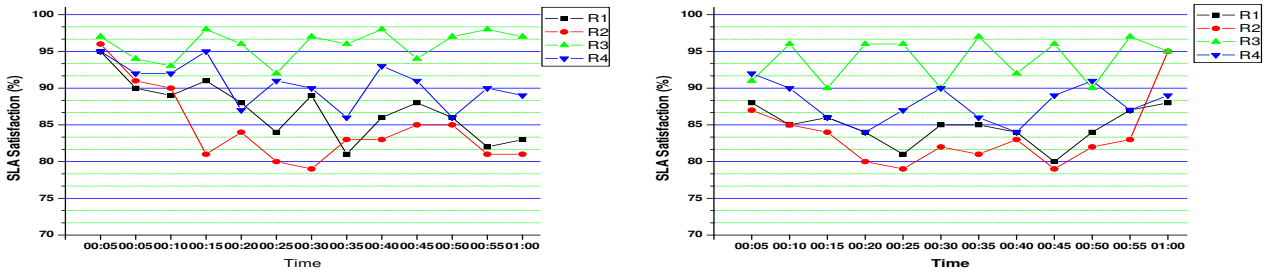
Fig. 9: The performance of the the adaptive management of the replication delay for the cloud-hosted database replicas.

different replicas. The results show that *us-west-2* is always the first location that add a new replica because it is the closest location to *us-west-1* which hosts the replica that firstly violates its defined SLA data freshness. The results also show that as the monitor interval increases, the triggering points for adding new replicas are usually delayed. On the contrary, the results of Figure (9c) and Figures (9g) to (9i) 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 replicas.

In general, the results of our experiments show that the adaptive replication controller can play an effective role on reducing the replication delay of the underlying replicas by adding new replicas when necessary. It is also observed that with more replicas 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 database replicas that can ensure low replication delay because it adds new replicas only when necessary based on the application-defined SLA of data freshness for the different underlying database replicas.

VIII. Provisioning The Database Tier Based on SLA of Transaction Response Times

Another consumer-centric SLA metric that we consider in our framework is the total execution times of database transactions (response time). In practice, this metric has a great impact on the user experience and thus satisfaction of the underlying services. In other words, individual users are generally more concerned about when their transaction will complete rather than how many transactions the system will be able to execute in a second (system throughput) [22]. To illustrate, assuming a transaction (T) with an associated SLA for its execution time (S) is presented to the system at time 0, if the system is able to finish the execution of the transaction at time ($t \leq S$) then the service provider has achieved his target otherwise if ($t > S$) then the transaction response cannot be delivered within the defined SLA and hence a penalty p is incurred. In practice, the SLA requirements can vary between the different types of application transactions (for example, a login application request may have an SLA of 100 ms execution time, a search request may have an SLA of 600 ms while a request of submitting an order information would have 1500 ms). Obviously, the variations in the SLA of different applications transactions is due to their different natures and their differences in the consumption behaviour of system resources (e.g. disk I/O, CPU time). In practice, each application transaction can send one or more operations to the underlying database system. Therefore, in our framework, consumer applications can define each transaction as pattern(s) of SQL commands where the



(a) Workload: 80/20 ((r/w)

(b) Workload: 50/50 ((r/w)

Fig. 10: Comparison of SLA-Based vs Resource-Based Database Provisioning Rules

transaction execution time is computed as the total execution time of these individual operations in the described pattern. Thus, the monitoring module is responsible for correlating the received database operations based on their sender in order to detect the transaction patterns [28]. Our framework also enables the consumer application to declaratively define application-specific *action rules* to adaptively scale out or scale in according to the monitored status of the response times of application transactions. For example, an application can define to scale out the underlying database tier if the average percentage of SLA violation for transactions T_1 and T_2 exceeds 10% (of the total number of T_1 and T_2 transactions) for a continuous period of more than 8 minutes. Similarly, the application can define to scale in the database tier if the average percentage of SLA violation for transactions T_1 and T_2 is less than 2% for a continuous period that is more than 8 minutes and the average number of concurrent users per database replica is less than 25.

We conducted our experiments with 4 different rules for achieving elasticity and dynamic provisioning for the database tier in the cloud. Two rules are defined based on the average CPU utilization of allocated virtual machines for the database server as follows: Scale out the database tier (add one more replica) when the average CPU utilization of the virtual machines exceeds of 75% for (**R1**) and 85% for (**R2**) over a continuous period of 5 minutes. Two other rules are defined based on the percentage of the SLA satisfaction of the workload transactions (the SLA values of the different transactions are defined as specified in the Cloudstone benchmark) as follows: Scale out the database tier when the percentage of SLA satisfaction is less than 97% for (**R3**) and 90% for (**R4**) over a continuous period of 5 minutes. Our evaluation metrics are the overall percentage of SLA satisfaction and the number of provisioned database replicas during the experimental time.

Figure 10 illustrates the results of running our experiments over a period of one hour for the 80/20 workload (Figure 10a) and the 50/50 workload (Figure 10b). In these figures, the *X-axis* represents the elapsed time of the experiment while the *Y-axis* represents the SLA salification of the application workload according to the different elasticity rules. In general, we see that, even for this relatively small deployment, SLA-based can show improved overall SLA satisfaction of different workloads of the application. The results show that the SLA-based rules (**R3** and **R4**) are, by design, more sensitive for achieving the SLA satisfaction and thus they react earlier than the resource-based rules. The resource-based rules (**R1** and **R2**) can accept a longer period SLA violations before taking any necessary action (CPU utilization reaches the defined limit). The benefits of SLA-based rules become clear with the workload increase (increasing the number of users during the experiment time). The gap between the SLA-based rules and SLA-based rules is smaller for the workload with the higher write ratio (50/50) due to the higher contention of CPU resources for the write operations and thus the conditions of the resource-based rules can be satisfied earlier.

Workload / Rule	R1	R2	R3	R4
80/20	4	3	5	5
50/50	5	4	7	6

TABLE II: Number of Provisioned Database Replicas

Table II shows the total number of provisioned database replicas using the different elasticity rules for the two different workloads. Clearly, while the SLA-based rules achieves better SLA satisfaction, they may also provision more database replicas. This trade-off shows that there is no clear winner between the two approaches and we can not favour one approach over the other. However, the declarative SLA-based approach empowers the cloud consumer with a more convenient and flexible mechanism for controlling and achieving their policies in dynamic environments such as the Cloud.

IX. Related Work

Several approaches have been proposed for dynamic provisioning of computing resources based on their effective utilization [29], [30], [31]. These approaches are mainly geared towards the perspective of cloud providers. Wood et. al. [29] have presented an approach for dynamic provisioning of virtual machines. They define a unique metric based on the data consumption of the three physical computing resources: CPU, network and memory to make the provisioning decision. Padala et.al. [31] 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 [14] 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 the application requirements. Rogers et al. [32] 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 (e.g., I/O, memory, CPU) consumption for each query. Floratou et al. [33] have studied the performance and cost 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. [13] introduced a virtualization design advisor that uses information about the database workloads to provide offline recommendations of workload-specific virtual machines configurations. To the best of our knowledge, our approach is the first that tackle the problem of dynamic provisioning the cloud resources of the database tier based on consumer-centric and application-defined SLA metrics.

A common feature to the different cloud offerings of the *platform storage services* and the *relational database services* 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 general, replicating for performance differs significantly from replicating for availability or fault tolerance. The distinction between the two situations is mainly reflected by the higher degree of replication, and as a consequence the need for supporting weak consistency when scalability is the motivating factor for replication.In addition, the *platform storage layer* and the *relational database as a service* are mainly designed for multi-tenancy environments and they are more centric to the perspective of the cloud provider. Therefore, they do not provide support for any flexible mechanisms for scaling a single-tenant system (consumer perspective).

X. Conclusions

In this paper, we presented the design and implementation details[10] of an end-to-end framework that facilitates adaptive and dynamic provisioning of the database tier of the software applications based on consumer-centric 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. The framework provides the consumer applications with declarative and flexible mechanisms for defining their specific requirements for fine-granular SLA metrics at the application level. The framework is database platform-agnostic, uses virtualization-based database replication mechanisms and requires zero source code changes of the cloud-hosted software applications.

References

[1] M. Armbrust and al., "Above the Clouds: A Berkeley View of Cloud Computing," University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, 2009.

[2] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance," *PVLDB*, vol. 3, no. 1, 2010.

[3] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC*, 2010.

[4] B. Suleiman, S. Sakr, R. Jeffrey, and A. Liu, "On understanding the economics and elasticity challenges of deploying business applications on public cloud infrastructure," *Internet Services and Applications*, 2012.

[5] S. Sakr, A. Liu, D. M. Batista, and M. Alomari, "A survey of large scale data management approaches in cloud environments," *IEEE Communications Surveys and Tutorials*, vol. 13, no. 3, 2011.

[6] W. Vogels, "Eventually consistent," *Queue*, vol. 6, pp. 14–19, October 2008. [Online]. Available: http://doi.acm.org/10.1145/1466443.1466448

[7] R. Cattell, "Scalable sql and nosql data stores," *SIGMOD Record*, vol. 39, no. 4, 2010.

[8] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective," in *CIDR*, 2011.

[9] D. Bermbach and S. Tai, "Eventual consistency: How soon is eventual?" in *MW4SOC*, 2011.

[10] D. J. Abadi, "Data management in the cloud: Limitations and opportunities," *IEEE Data Eng. Bull.*, vol. 32, no. 1, 2009.

[11] D. Agrawal and al., "Database Management as a Service: Challenges and Opportunities," in *ICDE*, 2009.

[12] S. Sakr, L. Zhao, H. Wada, and A. Liu, "CloudDB AutoAdmin: Towards a Truly Elastic Cloud-Based Data Store," in *ICWS*, 2011.

[13] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath, "Automatic virtual machine configuration for database workloads," in *SIGMOD Conference*, 2008.

[14] E. Cecchet, R. Singh, U. Sharma, and P. J. Shenoy, "Dolly: virtualization-driven database provisioning for the cloud," in *VEE*, 2011.

[15] P. Bodík, A. Fox, M. Franklin, M. Jordan, and D. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," in *SoCC*, 2010.

[16] D. Durkee, "Why cloud computing will never be free," *Commun. ACM*, vol. 53, no. 5, 2010.

[17] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *ACM Conference on Computer and Communications Security*, 2009.

[18] C. Plattner and G. Alonso, "Ganymed: Scalable Replication for Transactional Web Applications," in *Middleware*, 2004.

[19] A. Elmore and al., "Zephyr: live migration in shared nothing databases for elastic cloud platforms," in *SIGMOD*, 2011.

[20] Y. Wu and M. Zhao, "Performance modeling of virtual machine live migration," in *IEEE CLOUD*, 2011.

[21] E. Brewer, "Towards robust distributed systems," in *PODC*, 2000.

[22] D. Florescu and D. Kossmann, "Rethinking cost and performance of database systems," *SIGMOD Record*, vol. 38, no. 1, 2009.

[23] T. Kraska and al., "Consistency Rationing in the Cloud: Pay only when it matters," *PVLDB*, vol. 2, no. 1, 2009.

[24] K. Keeton, C. B. M. III, C. A. N. Soules, and A. C. Veitch, "LazyBase: freshness vs. performance in information management," *Operating Systems Review*, vol. 44, no. 1, 2010.

[25] L. Zhao, S. Sakr, A. Fekete, H. Wada, and A. Liu, "Application-managed database replication on virtualized cloud environments," in *ICDE Workshops*, 2012.

[26] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, S. Patil, A. Fox, and D. Patterson, "Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0," in *Proc. of Cloud Computing and Its Applications (CCA)*, 2008.

[27] L. Zhao, S. Sakr, and A. Liu, "Application-managed replication controller for cloud-hosted databases," in *IEEE Cloud*, 2012.

[28] S. Sakr and A. Liu, "SLA-Based and Consumer-Centric Dynamic Provisioning for Cloud Databases," in *IEEE Cloud*, 2012.

[29] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif, "Black-box and gray-box strategies for virtual machine migration," in *NSDI*, 2007.

[30] S. Cunha, J. M. Almeida, V. Almeida, and M. Santos, "Self-adaptive capacity management for multi-tier virtualized environments," in *Integrated Network Management*, 2007.

[31] P. Padala and al., "Adaptive control of virtualized resources in utility computing environments," in *EuroSys*, 2007.

[32] J. Rogers, O. Papaemmanouil, and U. Çetintemel, "A generic auto-provisioning framework for cloud databases," in *ICDE Workshops*, 2010.

[33] A. Floratou and al., "When free is not really free: What does it cost to run a database workload in the cloud?" in *TPCTC*, 2011.

[10] http://cdbslaautoadmin.sourceforge.net/