Performance and Cost Considerations for Providing Geo-Elasticity in Database Clouds

TIAN GUO, Worcester Polytechnic Institute
PRASHANT SHENOY, University of Massachusetts Amherst

Online applications that serve global workload have become a norm and those applications are experiencing not only temporal but also spatial workload variations. In addition, more applications are hosting their backend tiers separately for benefits such as ease of management. To provision for such applications, traditional elasticity approaches that only consider temporal workload dynamics and assume well-provisioned backends are insufficient. Instead, in this article, we propose a new type of provisioning mechanisms—geo-elasticity, by utilizing distributed clouds with different locations. Centered on this idea, we build a system called DBScale that tracks geographic variations in the workload to dynamically provision database replicas at different cloud locations across the globe. Our geo-elastic provisioning approach comprises a regression-based model that infers database query workload from spatially distributed front-end workload, a two-node open queueing network model that estimates the capacity of databases serving both CPU and I/O-intensive query workloads and greedy algorithms for selecting best cloud locations based on latency and cost. We implement a prototype of our DBScale system on Amazon EC2's distributed cloud. Our experiments with our prototype show up to a 66% improvement in response time when compared to local elasticity approaches.

CCS Concepts: • **Mathematics of computing → Statistical paradigms**; • **Information systems → Middleware for databases**; • **Computing methodologies → Modeling methodologies**; • **Networks →** *Cloud computing*;

Additional Key Words and Phrases: Distributed clouds, database elasticity, model-based provisioning

ACM Reference format:
Tian Guo and Prashant Shenoy. 2017. Performance and Cost Considerations for Providing Geo-Elasticity in Database Clouds. *ACM Trans. Auton. Adapt. Syst.* 12, 4, Article 19 (December 2017), 32 pages.
https://doi.org/10.1145/3095891

1 INTRODUCTION

Cloud platforms are increasingly popular for hosting web-based applications and services. Studies have shown that more than 4% of Alexa top million websites (He et al. 2013) are now hosted on cloud platforms and these contribute to more than 1% of the Internet traffic. Cloud platforms come in many flavors. Today's Infrastructure-as-a-service (IaaS) clouds support flexible allocation of server and storage resources to their customers using virtual machines (VMs). Recently,

This is an expanded and revised version of a preliminary paper that appeared at ICAC 2015 (Guo and Shenoy 2015). This work is supported by the National Science Foundation, under grant 1345300, grant 1229059 and grant 1422245. Manuscript was received August 16, 2016.
Authors' addresses: T. Guo, Computer Science Department, Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA, USA 01609; P. Shenoy, College of Information and Computer Sciences, University of Massachusetts, 140 Governors Drive, Amherst, MA 01003.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2017 ACM 1556-4665/2017/12-ART19 $15.00
https://doi.org/10.1145/3095891

Database-as-a-service (DBaaS) clouds have become popular as a method for hosting databases for cloud applications. In a DBaaS cloud, a customer leases a database from the cloud provider for storing and retrieving their data and offloads the tasks of managing and provisioning ("right-sizing") the database to DBaaS cloud provider. Since the application provider no longer needs to deal with the complexity of scaling their database to dynamic application workloads, DBaaS clouds simplify the task of building cloud applications. In such a scenario, a multi-tier web application is built by hosting the front-end tiers on servers leased from an IaaS cloud, while the back-end database tier of the application is hosted on a DBaaS cloud. A key benefit of IaaS and DBaaS cloud platforms is their ability to provide *elasticity*, where the cloud platform dynamically and autonomously scales the capacity allocated to the application or database tiers based on observed workload dynamics.

A concurrent trend is that today's cloud platforms are becoming increasingly distributed by supporting data centers in different geographic regions and continents. For instance, Amazon's EC2 and Microsoft's Azure offer a choice of 11 and 17 global locations, respectively, to their customers today. Distributed clouds are especially well suited for deploying cloud applications that service a geographically diverse workload. For such applications, network latency between end users and server replicas still plays an important role in affecting overall performance (Guo et al. 2015; Singla et al. 2014). Therefore, a distributed cloud with a large set of locations provides the flexibility to deploy application replicas so users can be serviced from the nearest cloud replica for the best performance. Studies (Xu et al. 2011) have shown that such *geo-distributed* applications see geo-dynamic workloads, where the workload sees both spatial and temporal fluctuations. Thus, in addition to well-known temporal fluctuations, such as time-of-day effects or seasonal fluctuations (Arlitt and Williamson 1997; Birke et al. 2012), the application sees *spatial* fluctuations where workload volume in one geographic region (e.g., North America) fluctuates independently of the workload volume seen from other regions (e.g., Asia or Europe).

However, existing elasticity mechanisms, in the form of autoscaling within a physical cloud location boundary, are not well suited for handling spatial fluctuations seen in today's geo-distributed applications. The limitations are mainly twofold. First, local elasticity mechanisms, when provisioning resources, are constrained to a single cloud location or a static subset of all available cloud locations. Second, current approaches are oblivious to the spatial workload dynamics associated with the geo-distributed applications. For instance, if an application that is deployed in two locations, say North America and Europe, sees a spatial increase in workload volume in Asia, current elasticity mechanisms will attempt to increase the provisioned capacity in the existing locations, whereas the proper response is to deploy new replicas in Asian cloud locations.

Instead, a different elasticity approach is needed that can handle both the temporal and spatial variations seen in today's geo-distributed applications' workload—we refer to such an approach as *geo-elasticity*. A geo-elasticity mechanism handles temporal changes by varying the provisioned capacity locally and handles spatial changes by provisioning replicas across regions and at new locations.

In this article, we explore the problem of designing a geo-elasticity mechanism for Database-as-a-service (DBaaS) clouds. DBaaS clouds are increasingly geo-replicated for reasons such as to provide better end-to-end user performances and high availability (DeCandia et al. 2007; Sovran et al. 2011; Corbett et al. 2012; Nawab et al. 2015). With such a mechanism in place, DBaaS provider can provision the right amount of DB servers in the best cloud locations to avoid violating service-level agreements (SLAs) between DBaaS provider and DBaaS customers—applications that host their backends using DBaaS clouds. Figure 1 presents a high-level illustration of how DBScale interacts with a multi-tier application that is deployed using both IaaS and DBaaS.

We identify four key challenges in designing geo-elasticity for DBaaS clouds. First, because application database tiers only see workload traffic from front-end tiers but do not handle end-client

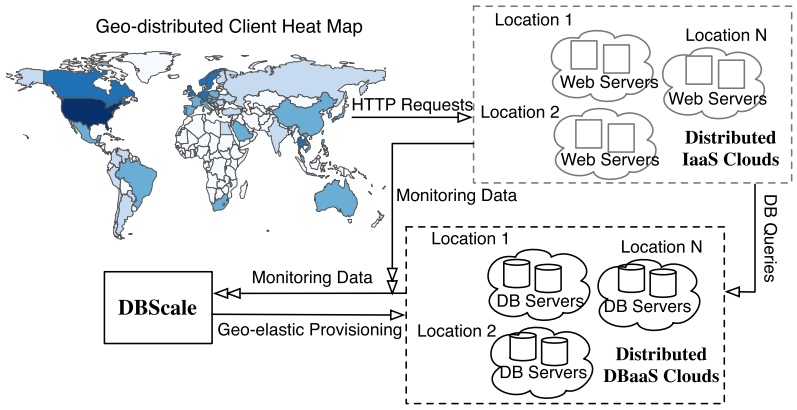


Fig. 1. **Illustration of using DBScale to manage a multi-tier application.** In this example, the multi-tier application serves different amounts of client workloads from different regions. The multi-tier application has its front-end web servers deployed in distributed IaaS clouds and its back-end database servers deployed in distributed DBaaS clouds.

traffic directly, inferring geographic workload distributions and associated spatial fluctuations for database servers is more challenging than for front-end tiers. Second, prior work on dynamic provisioning (Urgaonkar et al. 2005) for multi-tier applications usually make simplified assumptions about CPUs being bottleneck resources. Those approaches may not be well-suited for database tiers, because database can either be compute-intensive or I/O-intensive, or a mix of the two, depending on database query computational and I/O demands. Third, when a DBaaS cloud provisions database replicas, the task of maintaining consistency across replicas needs to be handled. Database consistency is a complicated task, especially in the presence of WAN replicas (Abadi 2012; Amir et al. 2003). In addition, consistency requirements are application-specific and therefore need to be handled differently for different applications. Finally, because end-to-end client performance depends on both front-end and back-end provisioning configurations, it is therefore very important to coordinate between IaaS and DBaaS to agree upon geo-elastic provisioning decisions and policies such as synchronizing provisioning completion time or using precopying.

Contributions. In this article, we address the above all four challenges with a model-driven middleware system called DBScale. DBScale implements an end-to-end solution that provides geo-elasticity for DBaaS clouds, from inferring dynamic database workloads of geo-distributed applications to provisioning database replicas in the *best* cloud locations. In designing and implementing DBScale, we make the following contributions:

- We propose a regression-based technique that uses the observed geographic distribution of the workload seen by the front-end tier to infer the resulting geographic distribution of the queries seen by the database tier. This regression model is used as the basis to predict future spatial workload for geo-elastic provisioning.
- We present a technique that models each database replica as a two-node open queueing network with feedback, with the CPU modeled as a M/G/1/PS queue and the disk modeled as a M/G/1/FCFS queue. In doing so, our model can effectively identify the resource bottlenecks that hinder the server response time and provide the basis for provisioning enough amount of servers without violating T^R_{SLA}, response time SLA.
- We analyze the performance and cost trade-offs of database workload assignment and propose two greedy algorithms that prioritize different objectives within the constraints of

network latency SLA, T^N_{SLA}. We also formulate an assignment problem using quadratic programming that minimizes operation cost.

- We implement a prototype of DBScale on Amazon EC2's distributed clouds and conduct detailed evaluations. Specifically, we run our experiments by injecting geo-distributed workloads from PlanetLab servers to a multi-tier application that are managed by DBScale in Amazon's distributed clouds. We compare the effectiveness of DBScale, in handling dynamic workload, to two other elasticity approaches—a local elasticity approach and a distributed caching approach. Our results show a 55% and a 36% improvement in mean response time when compared to local elasticity and the caching-based approach. In addition, we also evaluate our models and algorithms performance by comparing to benchmark measurements and through empirical data-driven simulations.

2 BACKGROUND AND PROBLEM STATEMENT

In this section, we first provide a background on distributed database clouds and then describe the application model assumed in our work and the specific problem of geo-elasticity in DBaaS clouds addressed in this work.

2.1 Distributed Database Clouds

Our work assumes a Database-as-a-service (DBaaS) cloud that allows application providers, or DBaaS customers, to lease one or more database tenants from the cloud platform. The DBaaS cloud provides SLAs on performance (e.g., response times) seen by the application and handles the task of configuring and provisioning sufficient capacity for each customer. Just as Infrastructure-as-a-service (IaaS) clouds support server instances of different sizes (e.g., small, medium or large servers), a database cloud also supports different types of database tenants. Small tenants, who have smaller storage and workload requirements, are hosted using a shared model where multiple small tenants share the resources of a single physical server. Large tenants, on the other hand, are hosted using a dedicated model, where each tenant is allocated all the resources of a physical server to support larger database or more-intensive workloads.

The DBaaS cloud itself can be implemented on top of a IaaS cloud where database tenants are housed in virtual machines ("server instances") of the IaaS cloud. While we assume such a virtualized environment for ease of prototyping, our approach could be easily generalized to non-virtualized setting. We assume that the DBaaS cloud is distributed and offers a choice of multiple locations to each application provider. Thus, in provisioning database tenants for DBaaS customers, DBaaS provider may choose a particular cloud location that is best suited for the application's needs or have a choice to a set of data center locations.

2.2 Application Model

Our work targets at multi-tier cloud-based applications that rely on DBaaS clouds for backend supports. Our targeted applications should have end-users that are spread across multiple geographic locations and hence service a geographically diverse workload. In addition, these applications not only experience temporal workload variations such as time-of-day effects, but also spatial variations, where the workload volume from different regions may vary independently (e.g., due to regional events or regional differences in the popularity of the application).

One prime example of our targeted applications is multi-tier web applications that consist of a front-end web tier and a backend database tier. We assume that the front tiers (HTTP and application tiers) are hosted on servers of a IaaS cloud, while the backend tier runs on a database in a DBaaS cloud. Both the IaaS cloud and DBaaS cloud have access to the same set of cloud

locations. Another emerging type of applications is mobile applications. Today's mobile applications are frequently designed with cloud backends to share and synchronize data among millions of app users and their multiple devices. Due to user mobility, spatial workload variations are execrated by these emerging mobile apps.

Further, such applications are designed to be replicable—that is, each tier can be scale horizontally by provisioning more replicas. Since the database tier is replicated, both within a particular cloud location and across locations, maintaining consistency of backend replicas is an important issue. We assume that consistency policies and mechanisms implemented by backend tier (and the DBaaS cloud) are dependent on the application's needs. In case of predominantly read-intensive database query workloads, such as those seen by databases hosting product catalogs of e-commerce store, a relaxed consistency technique may suffice, where the product catalogs replicas are updated periodically in batched mode. In other scenarios, where stricter consistency is desired, database replicas may need to be organized in a master-slave WAN configuration or a multi-master configuration[1]; these approaches will incur higher overheads, especially in WAN settings.

2.3 Geo-elasticity

Consider a distributed DBaaS cloud that hosts the database tier of geo-distributed multi-tier application as described above. The cloud platform is assumed to provide the primitives for scaling up or down the number of database replicas of customers' application within a single data center. However, given the geographically diverse workload, simply scaling the number of replicas at a given location is insufficient; the distributed DBaaS cloud needs to consider *where* workloads increase or decrease and decide how many replicas are needed at each cloud location. Such dynamic provisioning of capacity within and across cloud locations to handle both temporal and spatial workload fluctuations is referred to as *geo-elasticity*.

Our work focuses on providing geo-elasticity in a DBaaS cloud, while we do not assume any knowledge of provisioning mechanisms used by IaaS clouds that hosts front-end tiers. However, we assume a cooperative IaaS cloud that is able to incorporate provisioning decisions from a DBaaS cloud. Coordinating provisioning decisions between front-end and back-end tiers can be beneficial for good end-to-end user performance. Further, we also assume DBaaS customers specify their desired server types and our algorithm only considers provisioning additional servers of the same type.

2.3.1 The Need for Geo-Elasticity. A plausible alternative provisioning approach is called multi-site elasticity—a number of locations are statically chosen by the application developer and each site independently scales its front and back-end tiers based on the load seen at that data center. As we showed in our previous work (Guo et al. 2016), multi-site elasticity only provides suboptimal performance when the application starts seeing workload from a new region. This is because load is sent to a further data center until the application developer deploys a new replica at an additional data center. On the contrary, geo-elasticity will automatically detect this workload change and deploy a replica automatically at the new location. Moreover, as the number of data center locations grows—from $O(10)$ to $O(100)$ (Cisco Global Cloud Index 2016; Global Cloud Infrastructure 2016), it will be cost prohibitive to place a replica at every cloud location and optimal selection of locations for multi-site elasticity becomes ever more complicated. Therefore, it is better for the cloud platform to intelligently place replicas in cloud locations close to the end-users.

In summary, the ability for DBaaS cloud to dynamically varying and judiciously selecting the set of data center locations to host database replicas is not only critical for delivering performance

[1]Percona and EnterpriseDB both provide multi-master replication.

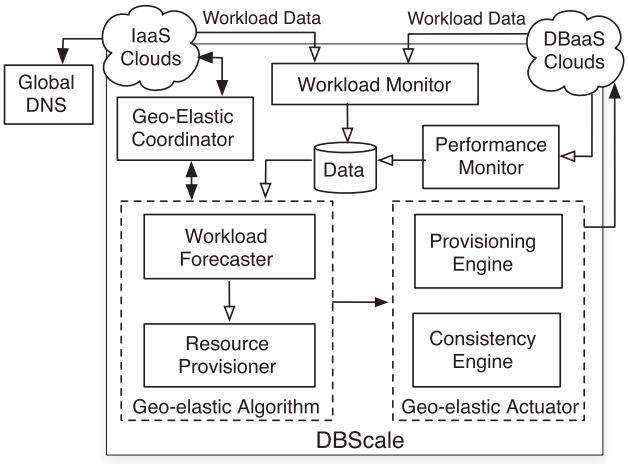


Fig. 2. **Key components of DBScale.** For simplicity, we only demonstrate the design and architecture of DBScale's central controller and omit light-weight daemons that report back workload and performance data from within DBaaS clouds. Here, arrows with solid head represent control decisions made by DBScale.

guarantee, that is, SLA, but also beneficial from cost perspective. Simply provisioning database replicas in all available data centers might not improve query response time but might even deteriorate it as shown in Section 8.5. This is because database replicas need to synchronize states among each other—introducing extra network delay to the response time.

2.4 Problem Statement

In this article, we are looking at the research problem of dynamically provisioning database replicas for multi-tier applications in distributed DBaaS clouds with minimal cost, without violating SLA.

Specifically, given a DBaaS cloud that has access to n data center locations L^k, and a cloud-based multi-tier application that serves client workload from m geographic regions L^c, we want to figure out the corresponding database workload dynamics, $\boldsymbol{\lambda} = [\lambda_1, \lambda_2, \ldots \lambda_m]$, for each provisioning period. Here, $\boldsymbol{\lambda}$ is a global database workload vector and each element λ_i represents peak query rates from ith client location.

After obtaining $\boldsymbol{\lambda}$, we want to assign those workload to cloud locations that satisfy our objective whether it is performance or cost, given network latency service level agreement, T^N_{SLA}. Generally speaking, we will have more client geographic regions than data center locations—that is $m > n$. Therefore, the above assigning process aggregates client workload into n cloud locations and yields workload $\boldsymbol{\omega} = [\omega_1, \omega_2, \ldots \omega_n]$, where ω_j represents workload that needs to be provisioned for jth cloud.

Then, we want to determine $\boldsymbol{D} = [d_1, d_2, \ldots d_n]$, number of database replicas to provision for each cloud location, based on server response time service level agreement T^R_{SLA}. Let d'_j denotes the number of database replicas already existing in cloud j, where $d'_j \leq 0$. By comparing d_j with d'_j for each cloud location j, we will decide to provision (or deactivated) $\mid d_j - d'_j \mid$ replicas.

2.5 Overview of Our Approach—DBScale

Figure 2 shows the high level design of DBScale that interacts with both IaaS clouds and DBaaS clouds. For completeness, we also depict a global DNS that should be notified whenever

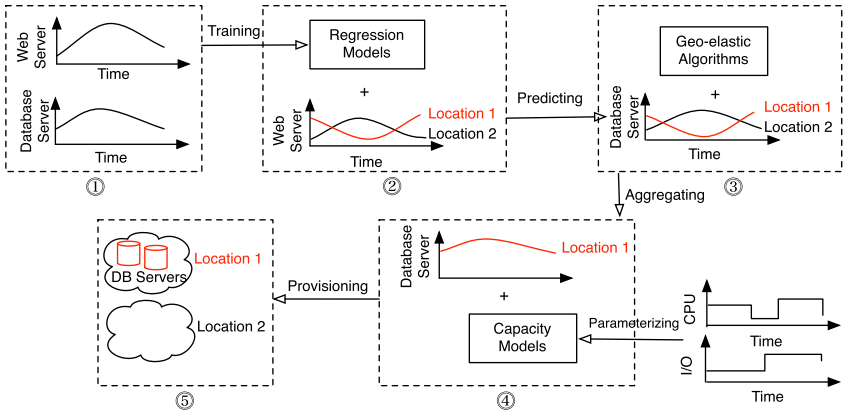
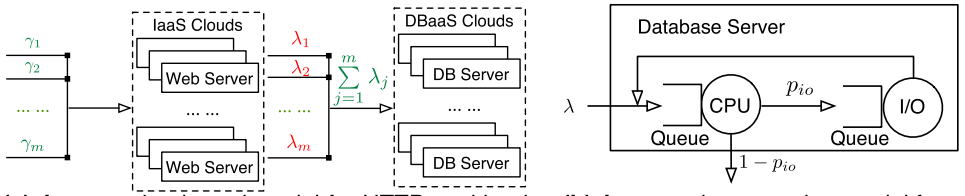


Fig. 3. **Typical workflow of DBScale.** We generalize DBScale's actions into four categories: training, predicting, aggregating, and provisioning. In this article, we take model-driven approaches using both regression and queueing models to estimate database workload and database server capacity. For choosing the best cloud locations, we use insights gained from linear programming formulation and a threshold-based greedy algorithm.

client-facing servers' IP addresses change. Note that such DNS lookup, with policies such as Latency Routing Policy (Amazon Route 53 2015), allows end user requests to be sent to the frontend replica that is hosted in the closest data center, from whom the database queries are sent to the closest replica. Therefore, we do not include time overhead associated with client-side DNS lookup in our SLA consideration. Similarly, frontend replicas also rely on DNS lookup to establish connections with the database replicas. However, slower DNS lookups only occur when there is no local database replicas inside the same data center as the frontend; and any potential performance overhead is amortized across long-lived database connections.

DBScale is responsible for dynamically provisioning databases in DBaaS clouds to handle temporal and spatial workload variations. Specifically, DBScale illustrated here is a logically controller that monitoring/analyzing global workload, deciding how to assign client workload to cloud locations, coordinating these decisions with IaaS provisioning engine, and provisioning/configuring database servers. However, our choice of centralized design does not have obvious impact on scalability. DBScale can easily scale up or out to support an increasing number of DBaaS customers. For each individual DBaaS customer, the time it takes for DBScale to make provisioning decision is affected by the number of actively used data centers. However, it is hardly going to be the bottleneck given the provisioning frequency, in the order of hours.

In Figure 3, we also provide a typical DBScale workflow to proactively provision database servers based on collected workload and performance data. For each provisioning period, DBScale performs the following four actions: *training*, *predicting*, *aggregating*, and *provisioning* in sequence to generate provisioning decisions for each DBaaS clouds customer. For the rest of design-related sections, we explain in detail based on the above workload about monitoring and predicting geo-dynamics database workload in Section 3; handling CPU-intensive and I/O-intensive database workloads with queueing-based capacity model in Section 4; figuring out where to provision database servers based on latency-first and cost-first greedy algorithms in Section 5 (and a Quadratic Programming formulation in Appendix A.1); and providing a step-by-step procedure to provide geo-elastic database clouds in Section 6. Implementation details of DBScale can be found in Section 7.



(a) A regression-based model for HTTP workload γ, and database workload, $\boldsymbol{\lambda}$.

(b) A two-nodes queueing-model for a single database server.

Fig. 4. **Model-driven geo-elastic approaches**. We model the relationship between front-end requests and database queries using regression-based models and leverage these models to predict temporal and spatial database workload. For estimating a single database server's capacity at a specific cloud location j we model the server as a two-node open queueing network.

3 GEO-DYNAMIC DATABASE WORKLOAD: WHERE AND HOW MUCH?

In this section, we look at how to obtain database workload dynamics $\boldsymbol{\lambda} = [\lambda_1, \lambda_2, \ldots \lambda_m]$ for each provisioning period. To do so, we need to be able to group queries to any one location j defined in L^c based on network proximity. However, queries are not directly associated with their originating clients—we do not know from which client location queries are generated. This is because client requests are first routed to web servers who then issue queries on behalf of clients. Therefore, the relationship between each query and its originating location is obscured by web servers.

One way to overcome this challenge is to define client regions L^c to be the same as IaaS cloud locations. Assuming IaaS has access to m' locations and clients are routed to the closest location among m' locations. By aggregating and analyzing database query logs from all active database servers, we can assign queries to IaaS cloud locations based on web server IP addresses. This way, even if we don't know the relationship between individual query and its originating client, we obtain a *coarse*-grained $\boldsymbol{\lambda}$ by using IaaS cloud locations as proxies—that is, database workload obtained using this approach is only reflecting spatial variations, if any, of front-end tier.

However, the effectiveness of the above approach depends largely on existing IaaS locations and whether IaaS employs geo-elastic provisioning. Currently, the number of IaaS cloud locations m' is in the low tens, and therefore they might not be representative for a global workload distribution. If IaaS clouds only provision using a subset of m' locations, then it will further reduce the usability of the above approach. That is, we will not be able to distinguish queries from Europe or U.S. East if IaaS only provisions web servers in U.S. East data center. In addition, this approach lacks the flexibility to produce $\boldsymbol{\lambda}$ for arbitrary client locations L^c, either in city, state, or country levels. Such flexibility can lead to *fine* granularity client workload information that can result in better workload assignment decision as described in Section 5.

Given the limitations of the above approach, we propose an effective regression-based approach that can produce database workload distribution $\boldsymbol{\lambda}$ with configurable precisions, using logs collected from both IaaS and DBaaS clouds.

3.1 Regression-Based Workload Prediction

In this section, we show how to obtain database query rate $\boldsymbol{\lambda}$ using a regression model that captures the relationship between λ_i and web request rate γ_i, with the help of $\sum_{j=1}^{n} \lambda_j$. In Figure 4(a), we show the interactions between variables that can be obtained directly using available logs (in green) and unknown variable (in red). Next, we first explain how to obtain $\boldsymbol{\gamma}$ and aggregate query rate $\sum_{j=1}^{n} \lambda_j$, and then introduce the regression model.

To obtain $\boldsymbol{\gamma}$, we first aggregate front-end request logs from all cloud locations; the request logs are assumed to include at least a time stamp and the end client's IP address. We then use an IP Geolocation technique[2] to determine the originating client location of each request. Given client locations L^c, all requests are then mapped to one and only one location in L^c that is closest. The use of IP Geolocation allows us to approximate network latencies between end clients and cloud locations with physical distances, and thus provides an intuitive approach to pinpoint the closest data center to end client as well as the flexibility to cluster end users with different granularities.

We count the number of requests that are mapped to each client location in L^c. For each client location i, we group requests by their time stamps and calculate the request intensity for specified time unit. At the end of this process, we will obtain $\boldsymbol{\gamma} = [\gamma_1, \gamma_2, \ldots \gamma_m]$, which represents the peak web workload from each client location i. Similarly, we can process all database logs and obtain database query rates from each IaaS cloud and subsequently the total query rates $\sum_{j=1}^{n} \lambda_j$.

For a specific application, the number of database queries triggered by front-end requests might vary depending on the types of requests. For example, a request to search for the best-selling products will have different database patterns than a request to finish placing order. Even so, it is still safe to assume that each front-end request will trigger *one or more* database query—we model this relationship with $\lambda = \alpha\gamma + \beta$, where α captures the linear relationship and β is an error term. Further, to capture the potential regional effect caused by different client workload pattern, we use linear models with different parameters (α_i, β_i) to model the relationship between front-end requests and the corresponding database queries from each client location:

$$\lambda_i = \alpha_i \gamma_i + \beta_i, \quad i = 1, 2 \ldots N. \tag{1}$$

Note, we can't obtain query rate λ_i by processing logs without knowing how clients are mapped to IaaS cloud locations. Therefore, we can't solve Equation (1) directly. However, relying on the fact that requests generated by all client locations are eventually contributing to the amount of database queries, we have

$$\begin{aligned}\sum_{j=1}^{n} \lambda_j &= \sum_{i=1}^{m} (\alpha_i \gamma_i + \beta_i) \\ &= \alpha_1\gamma_1 + \alpha_2\gamma_2 + \cdots + \alpha_m\gamma_m + \sum_{i=1}^{m} \beta_i \\ &= \alpha_1\gamma_1 + \alpha_2\gamma_2 + \cdots + \alpha_m\gamma_m + \beta.\end{aligned}$$

Here $\sum_{j=1}^{n} \lambda_j$ is the total database queries aggregated from all database replicas in L^k. For each provisioning period of length E, we prepare a data set $\{(\sum_j^n \lambda_j)_e, (\gamma_1)_e, (\gamma_2)_e, \ldots (\gamma_m)_e\}_{e=1}^{E}$ following above procedures. To find a model $(\alpha_1, \alpha_2, \ldots \alpha_m, \beta)$ that best explains these E data points, we use Least Squares Regression[3] to minimize the sum of squared residuals,

$$\min_{\alpha_1, \alpha_2, \ldots \alpha_m, \beta} \sum_{e=1}^{E} \epsilon_t^2, \tag{2}$$

where $\epsilon_t = \alpha_1\gamma_1 + \alpha_2\gamma_2 + \cdots + \alpha_m\gamma_m + \beta - \sum_j^n \lambda_j$.

[2]IP Geolocation is a technique that infers user's geographic location from IP address. We use MaxMind GeoIP2 (Maxmind GeoIP Service 2016) for this task.

[3]Other regression techniques could be applied here as well, such as robust linear model with Huber loss function or TukeyBiweight.

By solving Equation (2), we are only rewarded with a collective value β. To obtain $\{\beta_1, \beta_2, \ldots \beta_m\}$, we use a weighted function Equation (3) that distributes β to β_i based on corresponding workload portion—the more requests from a client location i, the more weight we assign to β_i,

$$\beta_i = \beta \frac{\gamma_i}{\sum_{i=1}^{m} \gamma_i}. \tag{3}$$

Combining Equations (2) and (3), we obtain m linear regression models (α_i, β_i) for each client location i and can use them to estimate the number of queries λ_i based on Equation (1). To be more specific, to predict the number of queries λ_i from location i at time $E+1$, we first take a series of M data points $[(\gamma_i)_{E-M}, (\gamma_i)_{E-M+1}, \ldots (\gamma_i)_E]$ and use ARIMA models (Box and Jenkins 1990), or any other standard time series prediction techniques, to predict $(\gamma_i)_{E+1}$. Then by substituting predicted front-end requests $(\gamma_i)_{E+1}$ into ith client location's regression model, we get $(\lambda_i)_{E+1} = \alpha_i(\gamma_i)_{E+1} + \beta_i$. We repeat the above steps for all m client locations and eventually obtain $\boldsymbol{(\lambda)_{E+1}} = [(\lambda_1)_{E+1}, (\lambda_2)_{E+1}, \ldots, (\lambda_m)_{E+1}]$, the database query distribution for time $E+1$.

4 PROVISIONING BASED ON SLA-BOUNDED DATABASE CAPACITY

To provision enough database servers to handle query workload, we need to have a way to estimate how many queries each database server can handle without violating service level agreement (SLA). One approach is gradually increasing realistic query workload intensity until server response time is violated. The maximum number of queries the database server can sustain is then to be used as server capacity. However, this approach might be less desirable, because it requires offline profiling and more importantly, it heavily relies on having perfect knowledge of database workload.

Therefore, we resort to queueing theory to estimate capacity online. In addition, in most scenarios, SLAs are specified as a high percentile of response-time distribution. Queuing theory also helps us reason about the tail behavior of the query service-time distribution. Specifically, by using queueing models, we can analyze the response-time distribution and obtain tail behavior and compare it with the pre-specified SLA to obtain server capacity.

4.1 Queueing-Based Capacity Estimation

Most prior work (Urgaonkar et al. 2005; N. Bennani and A. Menasce 2005; Villela et al. 2007) on using queueing-based models to perform dynamic resource provisioning only focus on estimating capacity for front-end tiers and assume CPU to be the bottleneck resource. Such approaches might not be ideal for database tiers, because databases may need to serve queries that are either CPU-intensive or I/O-intensive, or a mix of both. To account for both resource impacts' toward query response time, we present a database-specific queueing-based model that keeps track of both CPU and I/O utilizations. Note, we assume front-end servers will send queries directly to individual database servers—these database servers do not share a centralized queue and therefore are modeled individually.

Specifically, we model the database replica (on a dedicated host) as a two-node open queueing network with feedback, where the CPU is modeled as a M/G/1 processor sharing (M/G/1/PS) queue and the I/O device as a M/G/1 first come first serve (M/G/1/FCFS) queue as in Figure 4(b). Here, we model the arrivals of external queries as a Poisson process with average rate of λ. Immediately

following this, queries arrival at CPU and I/O also satisfy poisson distribution with λ_{cpu} and λ_{io},

$$\begin{cases} \lambda_{cpu} = \dfrac{\lambda}{1 - p_{io}} & \text{(4a)} \\ \lambda_{io} = \dfrac{p_{io}}{1 - p_{io}}\lambda. & \text{(4b)} \end{cases}$$

Here, a query arriving at a database server will first be added to the CPU queue. When the query departs from CPU, it will either leave the database server with probability $1 - p_{io}$ or continue its processing by joining I/O queue with probability p_{io}—that is, p_{io} is the query visit ratio to I/O. In a high level, a query might alternate between CPU and I/O multiple times before a response is generated.

By modeling queries going through both the CPU and I/O, this two-node queueing network is able to factor in both CPU and I/O's contributions in affecting query response time. Let us denote query response time using T, the mean response time $E[T]$ of database queries is then the sum of time spent in CPU and I/O, that is, $E[T] = E[T_{cpu}] + E[T_{io}]$. We use a recent result from a queueing literature (Boxma et al. 2005) that provides approximation for both $E[T_{cpu}]$ and $E[T_{io}]$,

$$\begin{cases} E[T_{cpu}] = \dfrac{\bar{s}_{cpu}}{(1 - p_{io})(1 - \rho_{cpu})} \\ E[T_{io}] = \dfrac{p_{io}}{1 - p_{io}}\left[\dfrac{\bar{s}_{io}}{1 - \rho_{io}} + \dfrac{p(\bar{s}_{io}^{(2)} - 2\bar{s}_{io}^2)}{2(1 - \rho_{io})}\lambda\right], \end{cases} \tag{5}$$

where $\bar{s}_{cpu}$ and $\bar{s}_{io}$ denote average service time of CPU and I/O; ρ_{cpu} and ρ_{io} denote average utilizations of CPU and I/O; $\bar{s}_{io}^{(2)}$ is the second moment of the service time distribution of I/O.

Now, given a pre-specified SLA between DBaaS and database customers in the form of 95th percentile response time T_{SLA}^R, we need to satisfy the constraint $\alpha_T(95) < T_{SLA}^R$. Here, $\alpha_T(95)$ denotes the 95th percentile of response time T. If we assume T satisfies an exponential distribution, we then have:

$$P(T \leq \alpha_T(95)) = 1 - e^{-\frac{1}{E[T]}\alpha_T(95)}, \tag{6}$$

based on the definition of cumulative distribution function. Therefore, we have $e^{-\frac{1}{E[T]}\alpha_T(95)} = 0.05$ and by taking ln of both sides,

$$\alpha_T(95) = \ln 20 E[X] \approx 3E[X]. \tag{7}$$

Given the relationship[4] Equation (7) and the SLA constraint, we then have $E[T_{cpu}] + E[T_{io}] < \frac{T_{SLA}^R}{3}$. By substituting Equation (5) into the previous inequality, we obtain an upper bound on the maximum query rate λ^c that can be handled by a *single* database server at a specific cloud location violating the SLA T_{SLA}^R:

$$\lambda^c \leq \frac{\frac{2T_{SLA}^R}{3}(1 - p)(1 - \rho_{io}) - 2\bar{s}_{cpu}\frac{1 - \rho_{io}}{1 - \rho_{cpu}} - 2p\bar{s}_{cpu}}{p^2(\bar{s}_{io}^{(2)} - 2\bar{s}_{io}^2)}, \tag{8}$$

[4]For a general distribution, we can use *Markov Inequality* to obtain $\alpha_T(95) \leq 20E[T]$ and follow the same steps to obtain a bound on α^c.

$$\lambda^c \leq \frac{\frac{SLA}{3}(1-p)(1-\rho_{io}) - 2\bar{x}_{cpu}\frac{1-\rho_{io}}{1-\rho_{cpu}} - 2p\bar{x}_{cpu}}{p^2(\bar{x}_{io}^{(2)} - 2\bar{x}_{io}^2)}. \quad (9)$$

4.2 Obtaining Model Parameters

Here, we explain how to obtain all model parameters for estimating $E[T]$ either by direct measurements or reasonable approximations. First, we need to empirically measure the CPU utilization, I/O utilization (using Linux tools such as sysstat), as well as per-query log that includes query timestamps and query execution time (by turning on MySQL slow logging and setting the `long_query_time` to 0 to record every query executed). We can directly estimate ρ_{cpu} from CPU utilization logs at a predefined time granularity, database query arrival rate λ by processing the per-query log and $\bar{s}_{io}$ and λ_{io} from the I/O utilization log. Based on Equation (4b), we can estimate p_{io} by substituting λ and λ_{io}. Since we do not have easy access to $\bar{s}_{cpu}$ and ρ_{cpu}, we approximate these two parameters using the Little's Law (Little 1961). Specifically, $\bar{s}_{cpu} = \rho_{cpu}\frac{1}{\lambda_{cpu}}$ and $\rho_{io} = \lambda_{io}\bar{s}_{io}$, where we obtain λ_{cpu} using Equation (4a). Note that these are overestimations due to extra logging overhead and resource interference—that is, we make conservative estimates of λ^c. Finally, to estimate database server capacity for a new datacenter location, we use the average of measured statistics across all available data centers as an initial approximation.

5 NETWORK SLA CONSTRAINED WORKLOAD ASSIGNMENT

In this section, we look at *where* to provision server resources for client workload $\boldsymbol{\lambda} = \{\lambda_1, \lambda_2, \ldots \lambda_m\}$ to satisfy T^N_{SLA}, 95th percentile network latency. Without loss of generality, we normalize $\boldsymbol{\lambda}$ with server capacity λ^c from Equation (9) and obtain a new normalized workload vector,

$$\lambda^N = \{\lambda^N_1, \lambda^N_2, \ldots \lambda^N_m\}, \quad \lambda^N_i = \frac{\lambda_i}{\lambda^c}. \quad (10)$$

Effectively, $\lceil \lambda^N_i \rceil$ represents the number of servers needed for client location i. Let us define χ_{ij} as the fraction of client workload λ_i that is assigned to and provisioned in cloud location j. Here, $\chi_{ij} \in [0, 1]$. An eligible assignment matrix $\chi_{m\times n}$ is one that satisfies T^N_{SLA}. Given the assignment matrix χ, we can express the normalized database workload for cloud location j,

$$\begin{aligned}\omega_j &= \chi_{1j}\lambda^N_1 + \chi_{2j}\lambda^N_2 \cdots + \chi_{mj}\lambda^N_m \\ &= \sum_{i=1}^{m} \chi_{ij}\lambda^N_i. \end{aligned} \quad (11)$$

Here, the number of servers that need to provision for cloud location j is then $\lceil \omega_j \rceil$. We define a cost and performance metrics as follows and use them as guidelines to evaluate the effectiveness of eligible assignments $\boldsymbol{\chi}$.

Cost Analysis. We consider three different cost aspects in calculating the operational expenditure (OPEX) of serving $\boldsymbol{\lambda}$ workload for the next provisioning period (V hours). Our cost analysis is based on current commercial cloud pricing models, specifically, we use Amazon's model as a concrete example. The first cost is hourly cost c^s for renting server resources. Therefore, the total rental cost is $C^s = c^s\lceil\omega\rceil V$, proportional to the number of servers and renting time.

The second cost we consider is data storage cost. The storage need at each server is defined as a continuous random variable D. We denote the database size at the beginning of each provisioning period as d^{DB} and the probability of inserting a new data entry as p^{ins}. The size of new data entry

is denoted with a continuous random variable U, and we assume knowledge of $\mathbb{E}[U]$. Therefore, the expected storage need of a server at v^{th} hour is $\mathbb{E}(D \mid V = v) = d^{DB} + \lambda^c p^{ins} \mathbb{E}[U]v$. Let us define c^d as the hourly cost for storing unit amount of data. Thus, the total storage cost across all servers is $C^d = \sum_{v=1}^{V} \mathbb{E}(D \mid V = v) \lceil \omega \rceil c^d$.

The last cost we consider is the cost for transferring data. We define a continuous random variable R to express the size of outbound Internet traffics for each server. The expected data transfer for vth hour in one provisioning period is $\mathbb{E}(R \mid V = v) = \lambda^c \mathbb{E}[U]v$. Let us define c^t as the hourly cost for transferring unit amount of data, we can then express the total transfer cost as $C^t = \sum_{v=1}^{V} \mathbb{E}(R \mid V = v) \lceil \omega \rceil c^t$. In all, by combining all three cost components, we have the formula to calculate cost to serve ω_j workload at cloud location j,

$$C(\omega_j) = C_j^s + C_j^d + C_j^t. \tag{12}$$

Performance Analysis. We are interested in analyzing the achieved network latency between client and database servers. To do so, we record latency values between communicating client and server and obtain the true network latency distribution. This type of application-level measurements provide good estimates for all individual latency pairs. However, it is excessive for our use cases and it might not always be feasible. Instead, we approximate the true distribution with $T^N = \{(A_{ij}, n_{ij}) \mid \forall i \in L^c, \forall j \in L^k\}$, where n_{ij} denotes the number of occurrences of latency A_{ij} between client location i and cloud location j. In essence, T^N is a multiset and each element A_{ij} has multiplicity n_{ij}. T^N is a reasonable approximation for targeted network latency, because client location i represents a cluster of clients from nearby geographic locations.

5.1 Greedy Algorithms

Next, we describe two greedy algorithms that focus on minimizing either network latency or provisioning cost.

Latency-first Greedy. We first sort all client locations L^c based on their workload intensity in descending order. For each client location j, we first find the set of eligible cloud locations $S_i = \{j \mid A_{ij} \in [0, T_{SLA}^N]\}$. That is, a cloud location j is said to be eligible for client location i if A_{ij}, the network distance between these two locations, is smaller than T_{SLA}^N. Note that $S_i \neq \emptyset, \forall i$ based on our assumption of T_{SLA}^N.

We then assign workload λ_i from client location i to the closest cloud location,

$$O_i^{lat} = \arg\min_{j \in S_i} A_{ij},$$

if there are enough resources. Otherwise, we move to the next closest cloud location in S_i until we successfully acquire an eligible cloud location. We repeat the above process for all client locations L^c and eventually reach a valid assignment χ^{lat}. For simplicity, we assume the total resources from all eligible cloud locations are sufficient to satisfy the demand of workload $\boldsymbol{\lambda}$. Based on χ^{lat}, we can express the aggregate workload for each cloud location and the associated cost and network latencies:

$$\omega_j^{lat} = \sum_{i}^{m} \lambda_i^N \mathbb{1}\{O_i^{lat} = j\}, \quad \forall j \in L^k, \tag{13}$$

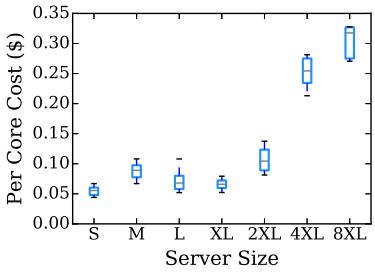
$$C^{lat} = \sum_{j=1}^{n} C\left(\omega_j^{lat}\right), \tag{14}$$

$$T_{lat}^N = \{(A_{iO_i^{lat}}, \lambda_i) \mid \forall i \in L^c\}. \tag{15}$$

EBS Storage Type(GB-month)	Max($)	Min($)	Std. Dev. ($)
General purpose SSD	0.19	0.10	0.03
Provisioned IOPS SSD	0.24	0.13	0.03
Throughput optimized HDD	0.09	0.05	0.01
Cold HDD	0.05	0.03	0.01
Snapshot (to S3)	0.13	0.10	0.01

EC2 Data Transfer (GB)	Max($)	Min($)	Std. Dev. ($)
Outbound Internet Traffics	0.25	0.09	0.05

(a) Storage and data transfer Costs.



(b) Server costs.

Fig. 5. We use Amazon's distributed clouds as a case study and analyze price differences exhibiting in different cloud locations for storage, data transferring, and servers. For example, one can save up to 24% in renting *4xlarge* server by choosing a cheaper data center.

Cost-first Greedy. Observe that we might get a cheaper assignment than *latency-first* algorithm by considering the cost differences between different cloud locations, as shown in Figure 5. Based on this observation, we propose the *cost-first greedy* algorithm that assigns client workload to the cheapest eligible cloud.

Instead of choosing the closest available cloud location, we choose the cheapest location $O_i^{cos} = \arg\min_{j \in S_i} C(\lambda_j)$. This yields a different assignment matrix χ^{cos} that is associated with

$$\omega_j^{cos} = \sum_i^m \lambda_i^N \mathbb{1}\{O_i^{cos} = j\}, \quad \forall j \in L^k, \tag{16}$$

$$C^{cos} = \sum_{j=1}^{n} C(\omega_j^{cos}), \tag{17}$$

$$T_{cos}^N = \{(A_{iO_i^{cos}}, \lambda_i) \mid \forall i \in L^c\}. \tag{18}$$

Discussions. Note neither greedy approaches, when choosing the cloud location for client workload, consider the existing client assignment. Thus, we might end up provision $n-1$ extra servers than we should have for workload $\boldsymbol{\lambda}$. This is because in the best case scenario, we only need to provision a total of $\lceil \sum_{i=1}^m \lambda_i^N \rceil$—that is, client workload is aggregated perfectly. While in the worse case scenario, we end up provision one extra server for every cloud location to handle $\omega_j - \lfloor \omega_j \rfloor$ fractional of workload.

To further reduce the cost overhead, we include a quadratic programming (QP) formulation in Appendix A.1. But in practice, QP formulation might not be desirable due to limited benefits and time complexity. Specifically, the potential cost saving is bounded by the cost of renting $n-1$ servers. In current pricing models, QP formulations do not have an effect on reducing either storage or bandwidth costs. When the number of cloud locations is reasonably small compared to provisioned servers, the saving of QP is negligible. As the number of cloud locations grows, the time complexity of solving this QP increases significantly. This makes it impractical as an online provisioning solution.

6 PUTTING IT TOGETHER

DBScale combines regression-based workload prediction, queueing-based capacity estimation, and greedy workload assignment algorithms to implement geo-elasticity for DBaaS, as summarized

below. DBScale periodically involves the following four steps, for example, every day, or when SLA s are violated.

Step 1: Where to provision? DBScale obtains database workload distribution $\boldsymbol{\lambda}$ from all m client locations using regression-based prediction model described in Section 3.1. Then, DBScale uses one of the algorithms from Section 5 based on customer's specification to generate workload assignment matrix $\boldsymbol{\chi}_{m\times n}$—each entry χ_{ij} specifies how much workload from client location i is to be assigned to cloud location j. Last, DBScale figures out workload to be provisioned for cloud locations, $\hat{\boldsymbol{\omega}} = \boldsymbol{\lambda}_{1\times m}\boldsymbol{\chi}_{m\times n}$. Those cloud locations with non-zero $\hat{\omega}_j$ are chosen for provisioning for the next period.

Step 2: How many resources to provision for each location? DBScale first parameterizes queueing-based capacity model, as described in Section 4, and then computes the maximum query rate λ^c that a single replica can handle without violating SLA T^R_{SLA}. Given the amount of query workload $\hat{\omega}_j$, a cloud location j is assigned for next provisioning period; DBScale then calculates the number of replicas $d_j = \lceil \frac{\hat{\omega}_j}{\lambda^c} \rceil$. If the number of replicas d_i differs from the value d'_i computed in the previous time interval (i.e., current provisioning), then $|d_i - d'_i|$ more replicas need to be provisioned or deactivated at this location. If $d'_i = 0$, then this indicates that location i has been newly chosen to provision database replicas.

Step 3: Coordinating with front-end tier. DBScale coordinates with front-end tier to enforce good end-to-end client performance through and especially after provisioning process. First, DBScale learns about the current configuration of front-end tier and any upcoming provisioning activities and uses such information to refine its provisioning policy. For example, if front-end tier decides to place a web replica at a new cloud location, DBScale needs to evaluate this decision in combination with its plan to make sure no SLAs are violated. Next, DBScale informs front-end tier about its provisioning plan and configuration such as whether snapshots are pre-copied. During provisioning, DBScale periodically updates front-end tier about its progress to synchronize provisioning completion time.

Step 4: How to provision database replicas? DBScale starts provisioning database replicas by first making a hot backup from an existing up-to-date replica using XtraBackup (Percona Xtrabackup 2015), a hot backup tool for MySQL database. If a full backup was created and archived in the destination clouds already, then DBScale will only request for an incremental backups that contains updated data. The hot backup tool produces a consistent point-in-time snapshot of the database without interrupting normal database processing at that replica. DBScale then transfers the snapshot to a DBaaS cloud server that will host the new replica. In the case where a new cloud location is chosen, the snapshot is transferred over WAN to this site. After transferring, DBScale uses the hot backup tool's crash recovery feature to load the snapshot into the database. DBScale supports two different modes to bring database replicas up-to-date, an offline approach and an online approach. If any updates are made to the database replica in the meantime, then DBScale uses an offline approach that acquires a read-lock on current replicas, fetches write queries and applies them to the newly provisioned replica(s). An alternate online approach is to make the new replica a slave and have it receive updates from an existing master (while this approach is suitable for master-salve configurations within a data center, doing so will incur higher overheads for master-slave configuration that run over WAN).

7 DBSCALE IMPLEMENTATION

7.1 Implementation Overview

We have designed and implemented DBScale as a middleware for managing geo-elasticity in DBaaS cloud. Our DBaaS cloud is built on top of Amazon EC2's distributed clouds that have tens of

cloud locations across the globe. To construct our DBaaS cloud, we first lease servers from distributed IaaS cloud and then run database replicas on those IaaS servers. Our prototype is based on the transactional MySQL database platform—that is, database tenants are provided as MySQL databases through the DBaaS cloud.

DBScale is implemented in `Python` and consists of two logical components. The high-level architecture of DBScale is shown in Figure 2. For simplicity, we only include the architecture of the central controller and omit showing daemons that run on each server inside DBaaS cloud. The central controller, by default, runs inside Amazon's U.S. East data center in Virginia. DBScale can also be configured to run its central controller in a new cloud location if it yields smaller communication overhead to all daemons.

Lightweight daemons that run inside all database servers are collecting required data, for example, workload and resource utilizations, and sending these statistics periodically to the central controller. Specifically, resource usage statistics at the database servers hosting the tenant replicas are measured using `sar` and `iostat` utilities, which yield the database server's CPU and I/O utilization. Note, the frequency of communicating with the central controller depends on whether a reactive threshold is triggered. If so, then daemons on these overloaded database servers will immediately notify the central controller. Otherwise, frequencies for each server are chosen uniformly from $[\psi, 10\psi]$ where ψ denotes the default provisioning frequency. If frequency is set at 10ψ, then daemons will contact the controller ten times within a provisioning window. This simple approach is used to avoid processing bottleneck in a single controller by effectively spreading processing workload into different time slots.

7.2 Implementation Details

Next, we describe implementation details for individual function modules of DBScale's central controller. These modules can be roughly divided into four interconnected pieces based on their functionalities. The interaction details between different modules can be found in Section 2.5.

First, *workload monitor* and *performance monitor* modules are responsible for gathering workload statistics and resource utilization from both IaaS and DBaaS clouds. We assume application developers who host databases in our DBaaS cloud expose APIs for DBScale to query the workload statistics of the front-end servers hosted in IaaS cloud. Using these APIs, we assume DBScale at least have access to aggregated requests per second for each cloud location. In an ideal scenario, workload monitor can gather web server logs directly from front-end replicas at each location and aggregates them, as discussed in Section 3, to analyze geographic distributions of client workload. DBScale can benefit from such fine granularity data and therefore make more informed provisioning and coordinating decisions.

Moreover, to collect data from DBaaS cloud, both *monitor* modules listen on a well-known port and collect data sent from daemons described above. Database workload information can be extracted from database query logs. Each query entry at least contains a query arrival time stamp, a requester's IP address, and a query execution time. Performance data can also be extracted from `CPU` and `I/O` logs, currently represented as averages over a 5s interval. All the workload and performance statistics are written to a SQLite database sequentially as they are processed. We use `ROWID` for primary key, and create additional three indexed columns, that is, data center location, server identification number and timestamp to represent each data point's attributes.

Next, *workload forecaster* and *resource provisioner* modules read data from SQLite database to construct a regression model and an ARIMA-based time-series model using Python's StatsModels library and to parameterize queueing models. These models are maintained and updated automatically based on data from a predefined historical time window. Currently, we set the historical time window for ARIMA model as one day and for the other two models as entire data history.

Such choices are only based on our limited experiences with benchmark experiments. It would be ideal to select window sizes for each model based on prediction accuracies. In addition, we also implement our two greedy algorithms in the *provisioner* that select cloud locations either based on network latencies to clients or server resource costs. Users can specify either *latency* or *cost* as a priority for assigning client workload. In all, DBScale combines these models and greedy algorithms to make geo-elastic provisioning plans periodically.

Then, *provisioning engine* takes a provisioning plan that specifies the number of different servers required for each cloud site and makes adjustment through Amazon EC2's APIs based on existing server resources. After each database server is up and running, we then use database hot backup tools to extract archived snapshots and load them into new replicas. Until now, provisioning engine has successfully prepared new database replicas, but these new database replicas might have obsolete data compared to the up-to-date database servers. The amount of such obsolete data depends on how many write requests have been committed since the snapshot is taken. Before handing them over to *consistency engine*, our *provisioning* module needs to contact one of up-to-date servers to fetch the committed transaction log and replay these transactions on new replicas. Afterwards, our *consistency* module chooses one of the two currently supported modes, that is, an offline batch mode and an online master-slave mode, depending on application's specific needs to synchronize new database servers. For example, when used for holding data such as product catalogs that see largely read queries, a simple batched update approach may suffice. Specifically, in the batch mode, we update all database replicas in a batch during the scheduled maintenance window. In the master-slave configuration, replicas are configured as either master or slave and write queries are only sent to the master who then relay to all the slaves. Our *consistency engine* picks databases inside a cloud site that has relatively low propagation delays to all other cloud sites, for example, a geographical central location, and configure them as master databases.

Last, *geo-elastic coordinator* acts as a bridge between IaaS and DBaaS, and notify both entities about any topology changes due to provisioning. The goal of our *coordinator* is to allow DBScale to make informed provisioning decisions, in conjunction with front-ends. As we show through a case scenario in Section 8.3.2, uncoordinated provisioning between IaaS and DBaaS might lead to undesired penalty spikes. To avoid scenarios such as running web and database servers in two cloud sites that are faraway, we incorporate a policy that always enforces provisioning database servers to "follow" the front-ends. And ideally, with cooperation from IaaS cloud, we can also synchronize the finish time of provisioning both tiers by delaying web server provisioning.

In summary, with the central controller taking care of different aspects of geo-elastic provisioning and distributed daemons collecting required data, DBScale thus proactively provisions database servers in accordance to the temporal and spatial client workload as well as front-ends' topology changes in the context of distributed clouds.

8 EXPERIMENTAL EVALUATION

We use end-to-end experiments and empirical-driven simulations to quantify DBScale's performance. First, we evaluate the efficacy of our models and algorithms. Then, we demonstrate performance improvement using geo-elasticity and compare DBScale to a caching-based approach. Last, we measure consistency overhead of provisioning database servers using DBScale.

8.1 Experimental Setup

Distributed Clouds: We use Amazon EC2's distributed cloud that spans more than ten global data center locations as the infrastructure support. We use EC2 to emulate IaaS clouds and create a DBaaS cloud by running MySQL database engines on rented IaaS servers. We use elastic block

stores (EBS) for hosting virtual machine images and database data. Further, we use Amazon's current pricing models (Figure 5 lists an exemplary summary) as a basis for our simulations.

Application appliance. We use TPC-W (The ObjectWeb TPC-W implementation 2005), a transactional web benchmark, as our multi-tier application. This java version of TPC-W consists of a front-end that runs on Apache Tomcat and a back-end database that runs on MySQL. We create separate appliances, virtual machine images, for its two tiers; unless specified otherwise, each database is configured with 10GB data. Both tiers of TPC-W are assumed to be replicable both within and across EC2 cloud locations. All the front-end VMs were running inside IaaS cloud and the back-end ones are running in DBaaS cloud. DBScale manages the replicas of each database tenant in DBaaS cloud and coordinates with an IaaS cloud manager.

Distributed Clients: We run the emulated clients on PlanetLab nodes that are globally distributed. We choose around 100 PlanetLab locations from North America, Europe, and Asia that were accessible at the time the experiments were performed. In our experiments, we use three workload mixes that represent different compositions of read and write requests, that is, default browsing and ordering workload from TPC-W, and a modified read-only browsing workload.

For each experiment run, we have TPC-W clients running on PlanetLab's nodes from different locations to send HTTP requests to the emulated online bookstore; requests are routed to the closest front-end replicas using a custom DNS-based load redirection. We warm up each replica for 5min before starting to collect data.

8.2 Geo-Elastic Models and Algorithms

In this section, we evaluate all the models and algorithms used by DBScale as a basis to provide geo-elasticity.

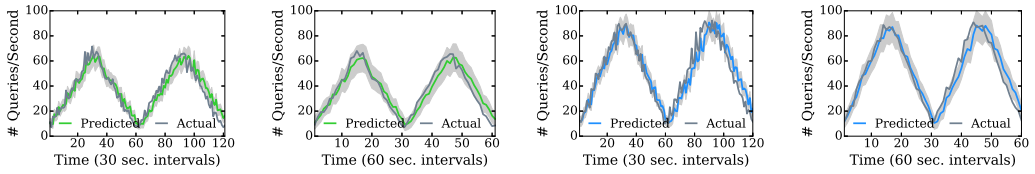
8.2.1 Regression Model Prediction. This experiment evaluates the effectiveness of the regression model proposed in Section 3.1 to predict database workload for distributed clients. We set up front-end web and back-end database servers in three cloud locations, that is, California, Virginia, and Ireland. For each cloud location, we then run TPC-W clients in a PlanetLab node that has a small network distance in terms of round trip time. Clients only send their requests to the preconfigured web and database pairs during the entire experiment that lasts for 1h. We control the number of concurrent clients from the same geographic location, therefore workload intensity, by assigning different starting time and transaction time to each client.

We repeat the process five times and use the data from the first four runs to train the regression model and obtain model parameter (α, β) for each client location. We then use the front-end requests from the fifth run as regression model input and calculate the predicted database query rates and reconstruct ground truth of database queries for each client location.

We plot the *Predicted* and *Actual* results for Pennsylvania clients with different workload mixes and prediction intervals in Figure 6. We observe that our regression model can make very reasonable predictions for both browsing (read-intensive) and ordering (read-write mix) workload with a mean error of 7.35%. Specifically, as we increase the prediction interval from 30s to 60s, our regression model makes better and smoother decisions.

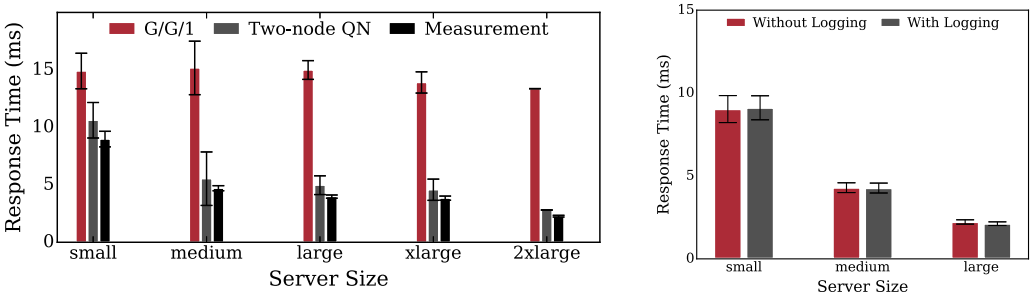
8.2.2 Queueing Model Prediction. Next, we evaluate our proposed database specific queueing model from Section 4.1 by comparing the estimated database server response times (which in turns yields server capacity) to a baseline G/G/1 queue (Guo et al. 2016) proposed in our previous work, and the empirical measurement.

We configure both front-end and back-end servers in Virginia data center and start a number of independent clients based on different server sizes. The independence between clients make sure query arrival to database server satisfies poisson process. And, we carefully control the number of



(a) Browsing: 30 secs. **(b)** Browsing: 60 secs. **(c)** Ordering: 30 secs. **(d)** Ordering: 60 secs.

Fig. 6. **Comparison of regression-based model predicted rates with empirical measurements.** Predicted and actual query rates over time for the browsing and ordering workload mixes. The shaded areas represent the 95th percentile confidence interval. For both workload types, the prediction accuracy is higher for a larger prediction window.



(a) Queueing model prediction accuracy. **(b)** Logging overhead.

Fig. 7. **Efficacy of the queueing model.** (a) For each database server size, we compare the empirically measured response times with predictions from a baseline G/G/1 model and our proposed database-specific queueing model. (b) We quantify the overhead for obtaining measurable variables for our two-node queueing network model.

clients, because we don't want to saturate the servers doing the test. For simplicity, we use a total of 100 clients for all server types that are under evaluation.

We collect both query logs and resource utilization logs from database servers for each experiment run that lasts half an hour. We then calculate the database server response time using both G/G/1 queue and our two-node queueing network (Equation (5)), and also obtain actual response time measurement by processing all the logs. We repeat the same process five times for each server type and calculate the average G/G/1 and DBScale estimations, the mean measurement value, and the 95th confidence interval across across all five runs.

We plot these comparisons for different server sizes in Figure 7(a). These results show that our database-specific queueing model has much better prediction accuracy when compared to the more general G/G/1 model. We also see that empirically measured response times lie within the 95% confidence intervals of our DBScale estimations, indicating a good prediction. Only in case of 2xlarge EC2 servers, where the empirical value is outside the 95% CI, we see a prediction error of 19%. In all cases, the model predictions are overestimates of the response times, indicating that the computed capacity will be conservative from a provisioning perspective. Further, in Figure 7(b), we quantify any potential performance impacts of measuring all variables for our two-node queueing networks. Our results show that logging these necessary has very minimal impact on database servers in turns of query response time, under reasonable server load. This indicates that our model is practical and can be used in an online fashion.

8.2.3 Geo-Distributed Workload Mapping Decisions. Last, we evaluate our two greedy algorithms' performance in tail network latency and daily operation cost by comparing them with a baseline algorithm. The baseline algorithm simply selects the cheapest cloud location and maps all client workload to that *single* location. Therefore, this baseline yields the lowest operation cost and a high tail network latency—but it does not provide any guarantee in satisfying T^N_{SLA}.

Our simulation is based on Amazon's current distributed clouds, which consists of 12 global cloud locations (Amazon Global Infrastructure 2016). Currently, end users might still experience hundreds of milliseconds network latency when connecting to the closest Amazon cloud location. However, public cloud infrastructures are going through rapid expansion with a total of 78 global regions as of year 2016 (Global Cloud Infrastructure 2016), and users from well-provisioned regions have as low as 33ms average network latency (Cisco Global Cloud Index 2016). With the predicted increase in the number of data centers (Cisco Global Cloud Index 2016), we envision that public clouds will be highly distributed and provide very low network latency to all end users in the near future. In fact, currently CDN providers are able to deliver static contents from a server that is less than 10ms away from major cities (Website Latency With and Without a Content Delivery Network 2016). Therefore, it is important to note results presented in this section can be improved significantly.

We collect empirical network latency traces by measuring the network distances between all PlanetLab nodes and Amazon clouds. This yields a latency matrix of size $(100, 12)$. We choose one particular server *4x large* server with optimized I/O and determine its capacity empirically. We use the above mentioned data as a basis for constructing simulation input.

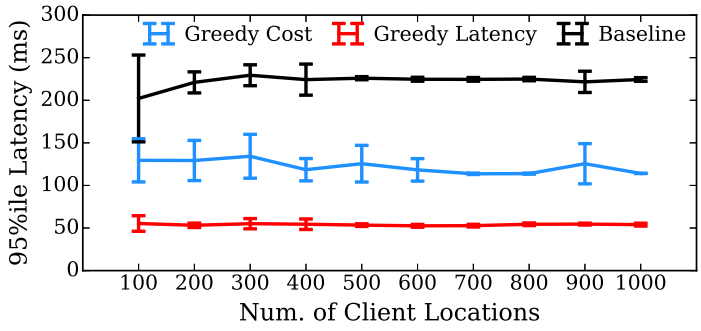
Specifically, for each simulation run, we configure the corresponding T^N_{SLA} and the number of client locations (as a proxy for client workload). We construct the set of client locations by uniformly selecting from PlanetLab nodes with replacement. We then generate normalized workload (compared to λ^c) associated with each client location by drawing a value from a uniform distribution with range $[0, 1]$. We vary simulation configurations and repeat each configuration for ten times and collect the network latency distribution and operation cost as defined in Section 5.

In Figure 8, we study how our two greedy algorithms behave with an increasing client workload and a fixed T^N_{SLA} of 200ms. Both greedy algorithms produce up to 172ms reduction in 95th percentile network latency. Latency-first greedy algorithm achieves optimal tail latency up to 80ms smaller than those of cost-first greedy algorithm. In addition, cost-first greedy algorithm achieves almost identical operation costs as with the baseline algorithm and an up to 10.6% saving when compared to latency-first greedy algorithm. This is mainly because cost-first algorithm can try to utilize eligible cloud location that is cheapest.

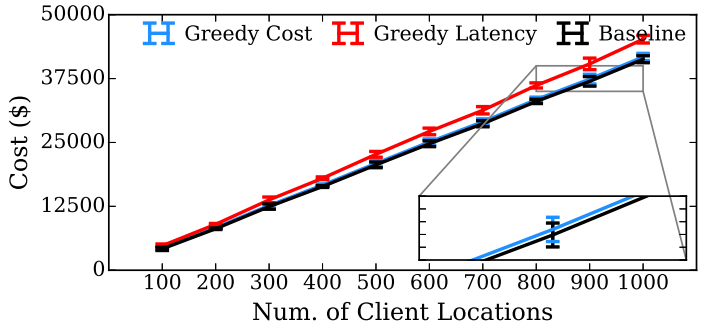
In Figure 9, we compare the performance of all three algorithms under different network SLA specification for assigning workload of one thousand client locations. We observe cost-first greedy algorithm behaves similarly to latency-first algorithm with a smaller T^N_{SLA} value, as shown in Figure 9(a). This is because the number of eligible cloud locations for each client location is determined by T^N_{SLA}—and when T^N_{SLA} is small enough, cost-first algorithm will pick the same cloud location as latency-first algorithm. Therefore, the tail latency performance of two greedy algorithms diverge with more relaxed network SLA values.

In summary, we show that cost-first greedy algorithm leads to higher cost savings when both client workload and network SLA increase while able to keep 95th percentile network latency within T^N_{SLA} specification. Because cost savings come from the ability to aggregate client workload and having access to more eligible cheaper clouds, we can expect higher savings in a more distributed cloud environment.

Conclusion: We empirically evaluate our regression-based workload prediction model, our queueing-based capacity model, and our workload mapping greedy algorithms using current distributed clouds



(a) 95^{th} percentile network latency.



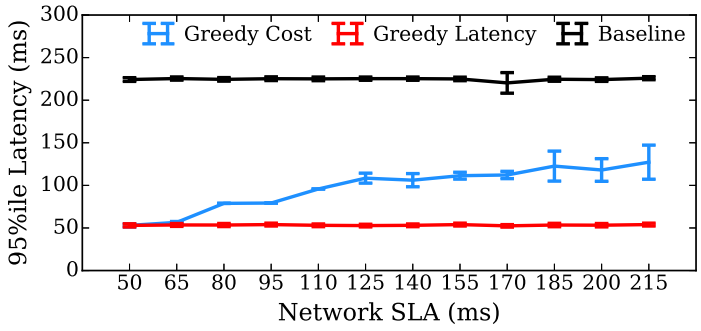
(b) Operation cost.

Fig. 8. **Client workload impact on workload mapping algorithms.** As the number of client locations increases, cost-first greedy algorithm can achieve 95th percentile network latency as low as 112 ms while saving up to 10.6% in operation cost.

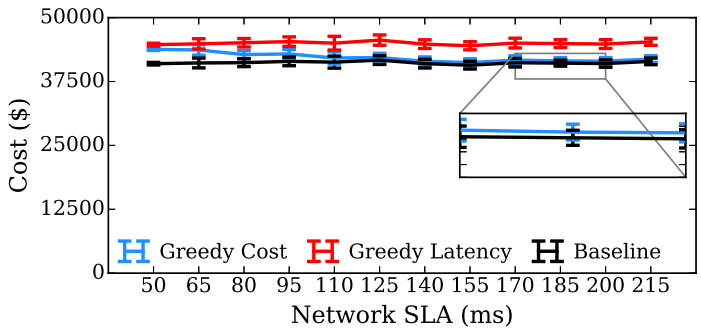
and geographically distributed clients. We show that our workload prediction only incurs a mean error of 7.35% and our queueing model produces reasonable overestimation when compared to empirical measurement. Further, our simulations demonstrate that our greedy algorithms can effectively make trade-offs between tail network latency and operation costs when compared to the baseline algorithm.

8.3 Benefits of Database Geo-Elasticity

In this section, we design a case study that demonstrates the potential performance improvement with geo-elastic provisioning. In addition, we compare the client performance of running geo-elasticity with four different policies. Figure 10 depicts our setup that involves two client locations, Pennsylvania and Germany, and two data center locations, Virginia and Ireland. Dark boxes represent web servers and light boxes represent database servers. For example, in the leftmost column, we have both clients from Pennsylvania and Germany make requests to a web server running inside Virginia's data center, who then fetches data from database running in the same data center. The top time axis shows the progress of different provisioning events with both local elasticity and geo-elasticity. The entire provisioning activity is broken down into three different phases, that is, starting provisioning, finishing provisioning web server, and finishing provisioning database server.



(a) 95^{th} percentile network latency.



(b) CDF of client response time.

Fig. 9. **Network SLA impact on workload mapping algorithms.** Because cost-first algorithm adjusts workload mapping decision by using Network SLA as a constraint, we observe the 95th percentile network latency increases accordingly. The cost differences between two algorithms are stable around 7.8% after network SLA is set to be larger than 125ms.

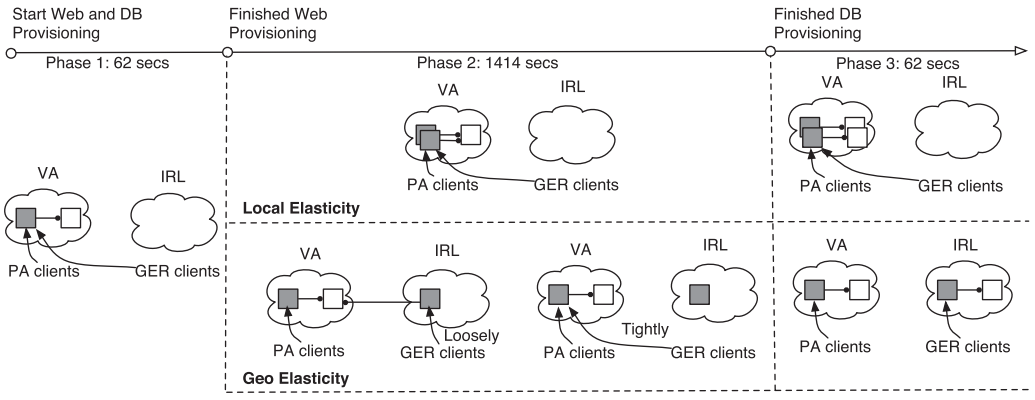
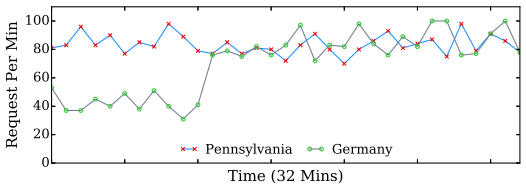


Fig. 10. **Illustration of elasticity mechanisms and provisioning policies.** We conduct an end-to-end experiment with different phases to demonstrate a policy-driven geo-elasticity is the most effective provisioning approach.

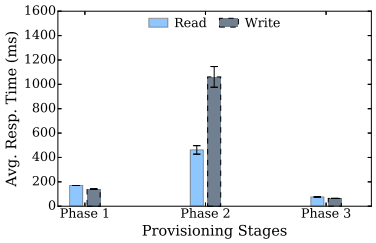


(a) Client request rates.

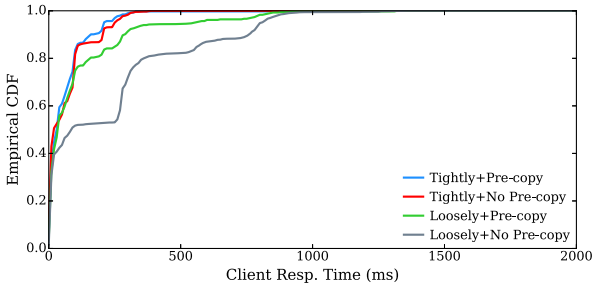
	Local Elasticity	Geo Elasticity
Mean(ms)	169	76
Std. Dev.(ms)	20	10
Conf. Int. (ms)	[167.07, 170.93]	[75.34 , 76.66]

(b) Response time statistics for local and geo elasticity.

Fig. 11. **Performance benefits for Geo-elasticity provisioning.** Geo-elasticity provides lower mean response times due to lower client-server network latencies.



(a) Mean Germany client response time.



(b) CDF of client response time.

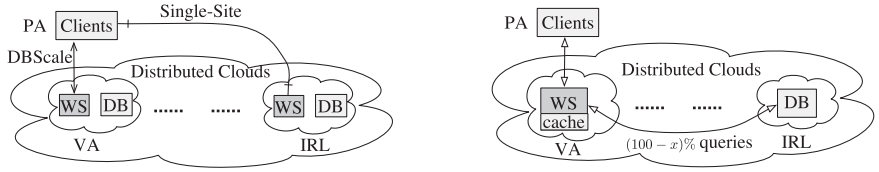
Fig. 12. **Performance benefits of tightly coupled provisioning and pre-copying.** A tightly coupled policy improves the 95th percentile of response time from 810ms to 250ms when compared to the loosely coupled policy. Pre-copying further improves 95th percentile of response time to 210ms.

8.3.1 Performance Improvement with Geo-Elasticity. We first compare the end-to-end performance improvement brought by provisioning in a geo-elastic way when compared to local elasticity. Figure 11(a) shows the average client requests for the entire experiment duration. We deliberately specify a very low server response time SLA—that is, at the end of the first provisioning epoch, our provisioning algorithms will scale up existing server resources. Note that, a choice of low SLA also eliminates potential performance deterioration caused by an overloaded server, making it easy to reason about the performance improvement.

To handle such a workload, local elasticity provisions additional servers locally, within the same Virginia data center location; while geo elasticity can provision capacity at any suitable cloud location, that is, Ireland cloud. The corresponding provisioning results are shown in Phase 3 of Figure 10. We record per request end-to-end response time for clients from both locations.

Figure 11(b) shows that geo-elasticity reduces average response time from 169ms to 76ms, an 55.03% improvement, when compared to local elasticity. The reason underlying the improvement is clients from Germany can now be fully served in the nearby Ireland cloud, instead of in the further Virginia data center. Therefore, all client requests from Germany are at least seeing an 70ms network round trip time reduction, from 100.3ms to 29.7ms.

8.3.2 Policy-Based Performance Improvement. Next, we scrutinize the end-to-end response time variations experienced by Germany clients when performing geo-elasticity with loosely-coupled policy. Figure 12(a) shows the response time spikes that Germany clients experience when provisioning activity is not synchronized between IaaS and DBaaS clouds—loosely coupled provisioning. During phase two, all client requests from Germany are first sent to the newly provisioned web server in Virginia who then makes query requests to the database server in Ireland. As a result,



(a) Baseline setups: DBScale and single-site elasticity. **(b)** Set up for $x\%$ cache hit rate.

Fig. 13. **Experimental setup for comparing DBScale to a caching approach.** The front-end tier is replicated and configured with an 1 GB in-memory cache in the caching approach. We use the same web and database server types for all the experiments.

requests that need to visit back-end servers multiple times to fetch desired data will experience 2× to 6× increase.

To reduce performance impact when provisioning for dependent resources, for example, front-end and back-end servers, we can either reduce provisioning duration external to clients or synchronize provisioning activities among resources. In the context of this case study, we can dramatically shorten database provisioning time from tens of minutes to a couple of minutes by pre-copying required data in advance; and we can enforce front-end servers to be configured to database servers within the same data center—data centers with acceptable network latency.

We plot client response time CDF obtained using four different policy combinations in Figure 12. Tightly coupled provisioning, with or without pre-copying, outperform loosely coupled provisioning with an up to 74% improvement for 95th percentile. When pre-copying the database snapshot to the destination Ireland cloud in advance, we only need to copy a delta of 100MB data during actual database provisioning. As a result, we drastically reduce duration of phase two and in turns cut down the number of requests that need to make transcontinental requests from Europe to U.S.A. With pre-copying enabled, loosely coupled provisioning yields up to 31% improvement for 95th percentile when compared without pre-copying.

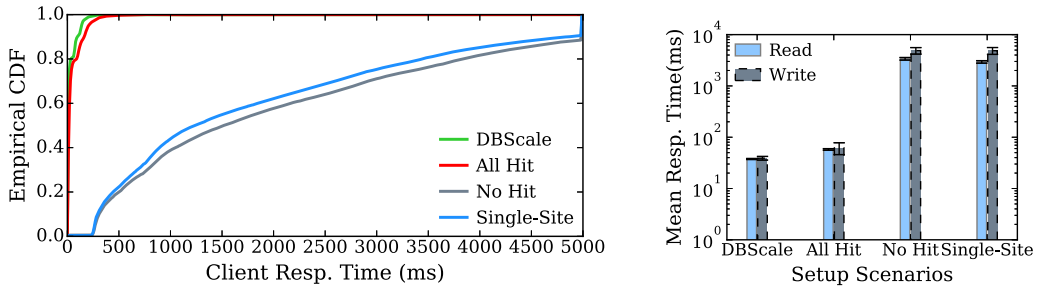
Conclusion: Geo-elasticity provisioning effectively reduces the mean end-to-end response time, thus improving performance for all clients. Tight-coupled and pre-copy policies are effective in reducing response time spikes during provisioning.

8.4 Comparing DBScale to a Caching Approach

In this section, we compare DBScale's performance to that of a caching-based approach. In our caching approach, the database server runs in a single *centralized* location while web servers are replicated in various geographic locations and use Memcached (Memcached 2015), or any other in-memory cache, to store recent query results.

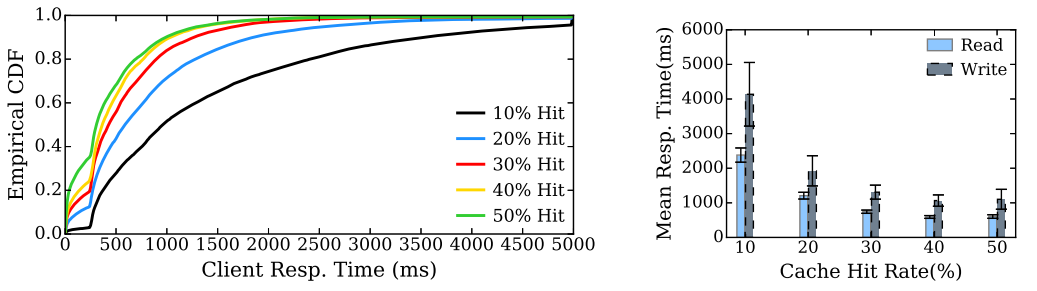
We modified TPC-W so read requests for data are sent to in-memory cache, who will then query remote database servers during cache miss. We use a small database of 512MB and allocate 1GB RAM for the in-memory cache. We warm up the in-memory cache using a modified read-only browsing workload mix. By warming up cache using known workload mix, we are able to control the desired cache hit rates. For example, if we set the cache hit rate to be 10%, each request will trigger a cache miss with a probability of 0.1 and be served directly from cache with 0.9 chance. The setup for this case study is illustrated in Figure 13. We run the default browsing workload mix (95% reads and 5% writes) from Pennsylvania clients for different setup and collect client end-to-end response time for all requests.

8.4.1 In-Memory Cache v.s. DBScale. We plot CDFs of end-user response time for four different scenarios, that is, DBScale, two extreme caching scenarios and single-site local elasticity, in



(a) CDF of response time. **(b)** Average response time.

Fig. 14. **CDF comparison of end-user response time of four different scenarios.** A caching approach with 100% hit rate has comparable performance to DBScale while a 0% hit rate causes performance to be similar to local single-site elasticity.



(a) CDF of response time. **(b)** Average response time v.s. hit rate.

Fig. 15. **CDF comparison of end-user response time with increasing hit rate.** As the hit rate increases from 10% to 50%, the 95th percentile response time improves by 72.18%, from 4780ms to 1330ms.

Figure 14. An *All hit* cache represents the best case scenario where all data requests can be satisfied from the local cache, while an *No hit* cache corresponds to the worse case scenario in which data are fetched from remote database across WAN. We show that DBScale has comparable performance with a perfect cache, because in both cases, front-end servers are able to fetch data locally, either from a local cache or a local database. In addition, we also demonstrate that both *Single-site* and a complete cold cache perform poorly, because all requests for data have to go to the further centralized database, either directly or from within the cache. Besides significant improvement of 95th percentile from 4.9s to 140ms, DBScale also reduces the mean response time, especially for write requests as shown in Figure 14(b) (log-scale y axis). In summary, DBScale behaves akin to the scenario where all requests are served from local cache, while single-site local elasticity is similar to a complete cache miss.

8.4.2 Impact of Cache Hit Rate. Next, we study performance impact with an increasing hit rate from 10% to 50%. In Figure 15(a), we plot the CDF of client response time and show that 95th percentile decreases from 478ms to 133ms when more requests are served from local cache. This is because the percentage of requests that avoid WAN latencies decreases as the hit rate increases from 10% to 50%. In addition, as shown in Figure 15(b), the benefits of caching only accumulate for predominantly read-intensive workloads. Requests that trigger update queries still need to visit the

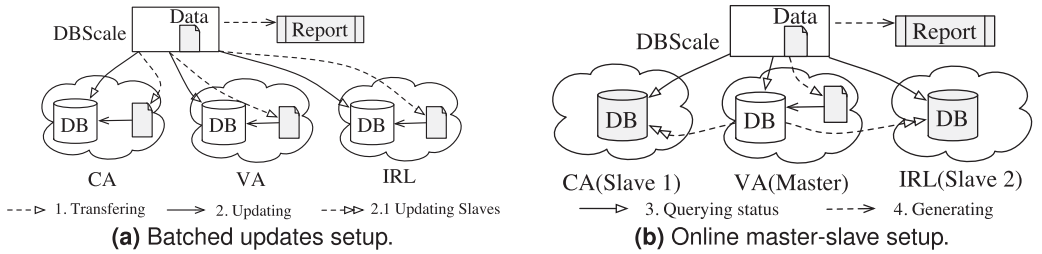


Fig. 16. **Experimental setup for updating databases in different locations.** Updates are first copied to all the cloud locations or the master database's location. Then, we either take the databases offline for batched update or configure a master-slave topology for online synchronization.

remote database server and therefore experience large network latency, resulting in poor average response time.

Conclusion: we demonstrate that a caching-based approach might provide comparable performance to DBScale, with high hit rate and a low fraction of write workload. However, in practice, the actual hit rate depends on a number of factors, such as the skew in query popularity distribution, cache size and replacement algorithms. DBScale does not depend on these factors and could yield good performance always (at the cost of needing consistency maintenance among replicas).

8.5 Consistency Maintenance Overheads

In our final set of experiments, we evaluate the overheads of maintaining consistency of database replicas that spread across transcontinental cloud locations. We compare two common approaches, that is, batched and online updates using MySQL master-slave configuration, for achieving database consistency. The experiment setup is illustrated in Figure 16 for both batch and online scenarios. And, we use TPC-W web application benchmark that is loaded with 13.43GB database.

8.5.1 Batched Updates Overhead. We measure the overhead of applying a varying amount of updates in batch mode during offline maintenance windows; the three database replicas are each hosted separately in Virginia, California and Ireland data centers, as shown in Figure 16(a). We measure the latency to apply updates at all replicas and restart all servers. Figure 17(a) shows the mean downtime for applying varying amount of updates across five runs along with the 95% confidence intervals. The figure shows that it takes 12.52min to update 1% of database data on a small server and as much as 60min to update 10% of the database on a medium server. In general, the downtime is cut in half as we move from a small server to medium or large servers. We observe the capacity differences and slightly different downtimes even for servers of same types in different cloud locations as shown in Figure 17(d). These results show that, barring under-sized small severs, batched updates can be a feasible option during maintenance windows, which themselves last for a few hours.

8.5.2 Online Master-Slave Maintenance. Finally, we study the overheads of using master-slave topology for executing database updates by measuring (i) the impact on maintenance time and (ii) the impact on foreground requests and client response time. As shown in Figure 16(b), we configure the database in Virginia as the master database and the other two as *Slave 1* and *Slave 2* in California and Ireland, respectively. Read queries are sent to the databases in the vicinity while write queries are sent to master database. We record the time to update 1% of database data as well as the end-users response time using this topology; all the database servers run on medium-sized servers.

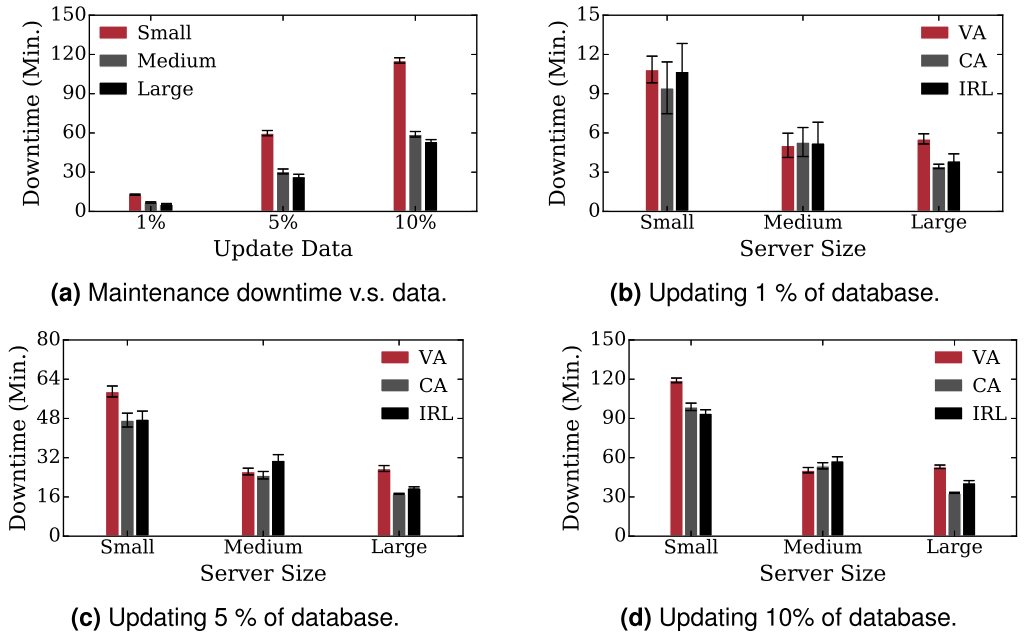


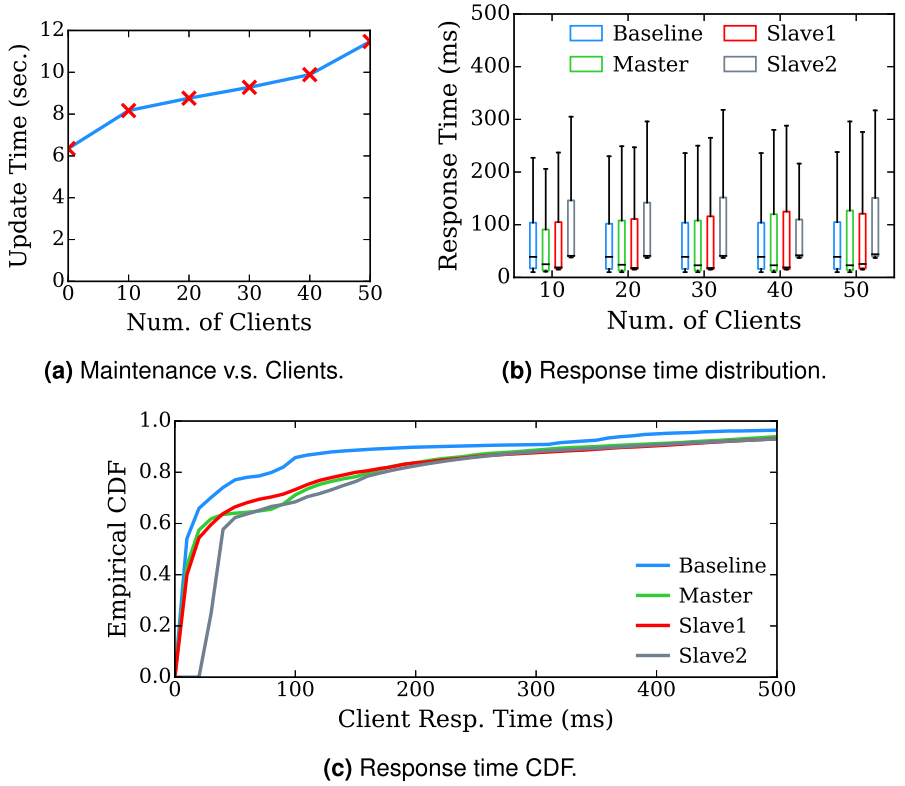
Fig. 17. **Batched updates Overheads.** The maintenance downtime (ranging from a few minutes to hours) due to batched updates is impacted by the amount of the data that need to be updated, and the server capacity, that is, server size and the cloud location.

Figure 18(a) shows that it takes 6.35min to update 1% data and as the front-end workload increase, the online update time increase too. Our observation suggests that to reduce the length of update time, that is, the impact duration on end-users, we could adjust the database server size based on end-users' workload during the online maintenance phase. To demonstrate the online maintenance activities' impacts on the end-users' response time, in Figure 18(b), we compare the client response time distribution of master and slaves compared to baseline *no writes* scenario for different levels of workload intensity. We observe no obvious impact on client response time distribution of master-slave updates approaches at different workload intensity, making it a feasible solution as well. Specifically, in Figure 18(c), we show that the 95th percentile response time increases from 400ms to 560ms for master and to 595ms for slaves for a 50-clients workload at each location.

Conclusion: we measure performance overhead of two consistency models provided in DBScale in varying scenarios. These measurements can serve as a guideline for configuring consistency for different application needs. We show the batch update time varies according to the amount of new data and server capacity. In master-slave mode, we show overhead increases with the client workload, with an up to 40% increase at 95th percentile response time at master.

9 RELATED WORK

Distributed cloud platforms have became a popular paradigm for hosting web applications. Their pay-as-you-go pricing model and flexible resource allocation make them well-suited for hosting applications with dynamic workloads (Arlitt and Williamson 1997; Birke et al. 2012). When physical resources are shared among multiple VMs, it becomes challenging to accurately model the resource usages for each VM (Kundu et al. 2010; Wood et al. 2008; Cherkasova and Gardner



(a) Maintenance v.s. Clients.

(b) Response time distribution.

(c) Response time CDF.

Fig. 18. **Impacts of online master-slave on update time and response time distribution.** As the workload of end-users increase, we observe a corresponding increase in the update latency. Also, response time CDF of both master and slaves behaves similarly to the *no-writes* baseline scenario.

2005) mainly due to interference of co-located VMs (Nathuji et al. 2010; Chiang and Huang 2011; Kambadur et al. 2012; Zhu and Tung 2012). The problem becomes more noticeable for applications with bursty workload characteristics (Mi et al. 2008). To overcome this hurdle, recent efforts have attempted to mitigate the impact of interference either by combining the VMs workloads (Meng et al. 2010) or by employing a novel performance prediction model that is capable of dealing with bursty workloads or even flash crowds (Casale et al. 2012). In our work, we combine our empirically measured distributed front-end workload and a regression-based model to predict the spatial and temporal variations in the backend database workload.

Queueing-based models have been used extensively to model cloud-based applications (Urgaonkar et al. 2005; Villela et al. 2007; N. Bennani and A. Menasce 2005), but most have focused on front-end servers. To parameterize those proposed models, it often requires performing empirical measurements on real system with predefined workload. However, due to potential costs of intrusive measurements and the volatility of workload mix, an alternative regression model (Zhang et al. 2007) was proposed to approximate the CPU demand with different transaction mixes to effectively model complex live systems with very few parameters. In our work, we focus on dynamic provisioning in distributed database cloud and model the database server as a two-node queueing network with feedback to track both CPU and I/O utilization.

As more database management tasks (Curino et al. 2011b; Popa et al. 2011) are offloaded to the cloud, researchers have begun to focus on adaptive and dynamic provisioning of database servers based on SLA (Xiong et al. 2011; Mozafari et al. 2013; Sakr and Liu 2012; Cecchet et al. 2011; Shankaranarayanan et al. 2014). These efforts on database provisioning include using models and tools to predict resource utilization and performance for OLTP databases (Mozafari et al. 2013; Curino et al. 2011a), cloning techniques to spawn database replicas (Cecchet et al. 2011; Nguyen et al. 2013), live migration techniques to horizontally scale up database server (Elmore et al. 2011), middleware approach to coordinate cloud-hosted applications and databases without violating SLA (Sakr and Liu 2012) and utilizing distributed cloud platforms for performance-aware data replication (Agarwal et al. 2010; Shankaranarayanan et al. 2014; Ping et al. 2011; Liu et al. 2013). Our focus here is on geo-elasticity, which is less well studied, and we propose the DBScale framework to handle geo-elasticity for cloud-hosted databases.

10 CONCLUSIONS

We proposed a new dynamic provisioning algorithm, called geo-elasticity, for DBaaS clouds to handle both temporal and spatial workload dynamics. Our work is motivated by the emergence of distributed clouds, the popularity of geographically distributed applications, and the paradigm for applications to host their back-end tiers in DBaaS clouds. To achieve geo-elasticity, we presented a regression-based prediction model that infers geographical workload distribution for database tier, and a two-node open queueing network model that estimates database capacity. Further, we proposed the geo-elastic algorithm that combines both models and two greedy workload assignment algorithms for provisioning database servers in distributed clouds.

We implemented a prototype called DBScale as a middleware based on Amazon distributed clouds and conducted comprehensive evaluations to quantify DBScale's performance. Specifically, we performed both end-to-end experiments as well as benchmark experiments to demonstrate the efficacy of our models, algorithms and DBScale as a whole. Our results showed up to a 66% improvement in response time when compared to local elasticity approaches. As part of future work, we plan to explore the benefits of provisioning using heterogenous cloud resources and considering cloud performance interference explicitly.

APPENDIX

A.1 Geo-Elastic Provisioning with Quadratic Programming

Both χ^{lat} and χ^{cos}, obtained through our greedy algorithms in Section 5.1, are binary matrices. That is, χ_{ij} is binary—either all or none of workload from client location i is assigned to cloud location j. In this section, we formulate the workload assignment problem using quadratic programming that allows us to assign client workload from the same location i to multiple cloud locations. This provides us flexibility to *split and pool* client workload and potentially reduce the total number of servers needed.

In a high level, this quadratic formulation reduces cost by finding cheapest cloud locations and aggregating client workload into as fewer servers as possible. In other words, for a subset client workload scenarios, our *cost-first* greedy algorithm will produce the assignments with the same costs,

$$\min \quad \sum_{j=1}^{n} \lceil \omega_j \rceil C_j(\omega_j), \tag{19}$$

subject to

$$0 \leq \chi_{ij} \leq 1, \quad \forall i \in L^c, \forall j \in L^k, \tag{20}$$

$$\sum_{j=1}^{n} \chi_{ij} = 1, \quad \forall i \in L^c, \tag{21}$$

$$\alpha_{95}(T_c^N) \leq T_{SLA}^N, \tag{22}$$

$$\omega_j \leq R_j, \quad \forall j \in L^k. \tag{23}$$

Recall that $C_j(\omega_j)$ Equation (12) is a function of ω_j Equation (11) and represents the total cost to serve ω_j workload at cloud location j. The objective function Equation (19) tries to minimize the total cost for all workload $\boldsymbol{\omega}$, where $\lceil \omega_j \rceil$ is the number of servers needed at cloud location j. Constraints Equations (20) and (21) makes sure that all client workload is assigned, and constraint Equations (22) and (23) ensure that network SLA and available resource is not violated.

By solving the above QP formulation, we obtain the assignment matrix $\boldsymbol{\chi}_{\boldsymbol{m \times n}}^{\boldsymbol{QP}}$. Combining with λ^N Equation (10), we get

$$\omega^{QP} = \lambda^N \chi^{QP}. \tag{24}$$

The total cost associated is then $C^{QP} = \sum_{j=1}^{n} \lceil \omega_j^{QP} \rceil C_j(\omega_j^{QP})$. Last, we use $T_{QP}^N = \{(A_{ij}, \lceil \chi_{ij} \lambda_i \rceil) \mid \forall i \in L^c, \forall j \in L^k\}$ to approximate the true network latency distribution. Note T_{QP}^N includes at most mn more data samples compared to actual measurement. But given a reasonable workload $\boldsymbol{\lambda}$, the extra samples will not affect statistics we are interested in, that is, mean and 95th percentile.

REFERENCES

Daniel Abadi. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* 45, 2 (Feb. 2012), 37–42. DOI: http://dx.doi.org/10.1109/MC.2012.33

Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. 2010. Volley: Automated data placement for geo-distributed cloud services. In *Proceedings of the Conference on Networked Systems Design and Implementation (NSDI'10)*. 17–32.

Amazon Global Infrastructure 2016. Amazon Global Infrastructure. Retrieved from http://aws.amazon.com/about-aws/global-infrastructure/.

Amazon Route 53 2015. Amazon Route 53: Choosing a Routing Policy. Retrieved from http://docs.aws.amazon.com/Route53/latest/DeveloperGuide/routing-policy.html.

Yair Amir, Claudiu Danilov, Michal Miskin-Amir, Jonathan Stanton, and Ciprian Tutu. 2003. *On the Performance of Consistent Wide-area Database Replication*. Technical Report.

Martin F. Arlitt and Carey L. Williamson. 1997. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on Networking* 5, 5 (1997), 631–645.

Robert Birke, Lydia Y. Chen, and Evgenia Smirni. 2012. Usage patterns in multi-tenant data centers: A temporal perspective. In *Proceedings of the 2015 IEEE International Conference on Autonomic Computing (ICAC'12)*.

George Edward Pelham Box and Gwilym Jenkins. 1990. *Time Series Analysis, Forecasting and Control*.

O. J. Boxma, R. D. van der Mei, J. A. C. Resing, and K. M. C. van Wingerden. 2005. Sojourn time approximations in a two-node queueing network. In *Proceedings of the International Trade Commission (ITC'05)*.

G. Casale, Ningfang Mi, L. Cherkasova, and E. Smirni. 2012. Dealing with burstiness in multi-tier applications: Models and their parameterization. *IEEE Transactions on Software Engineering* 38, 5 (2012), 1040–1053.

Emmanuel Cecchet, Rahul Singh, Upendra Sharma, and Prashant Shenoy. 2011. Dolly: Virtualization-driven database provisioning for the cloud. In *Proceedings of the Conference on Virtual Execution Environments (VEE'11)*.

Ludmila Cherkasova and Rob Gardner. 2005. Measuring CPU overhead for I/O processing in the xen virtual machine monitor. In *Proceedings of the Annual Technical Conference (ATEC'05)*.

Ron C. Chiang and H. Howie Huang. 2011. TRACON: Interference-aware scheduling for data-intensive applications in virtualized environments. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 47.

Cisco Global Cloud Index 2016. Cisco Global Cloud Index:Forecast and Methodology, 2015–2020. Retrieved from http://www.cisco.com/c/dam/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.pdf.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, 251–264. Retrieved from http://dl.acm.org/citation.cfm?id=2387880.2387905.

Carlo Curino, Evan P. C. Jones, Samuel Madden, and Hari Balakrishnan. 2011a. Workload-aware database monitoring and consolidation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'11)*.

Carlo Curino, Evan Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. 2011b. Relational cloud: A database-as-a-service for the cloud. In *proceedings of the 5th Biennial Conference on Innovative Data Systems Research*.

G. DeCandia, D. Hastorun, and M. Jampani. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operat.* (2007).

Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'11)*.

Global Cloud Infrastructure 2016. Regions Beyond Regions: Global Cloud Infrastructure Expansions. Retrieved from https://blog.fugue.co/2016-04-12-regions-beyond-regions-global-cloud-infrastructure-expansions.html.

Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. 2015. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM'15)*. ACM, New York, NY, 139–152. DOI: http://dx.doi.org/10.1145/2785956.2787496

T. Guo and P. Shenoy. 2015. Model-driven geo-elasticity in database clouds. In *Proceedings of the 2015 IEEE International Conference on Autonomic Computing (ICAC'15)*.

Tian Guo, Prashant Shenoy, and Hakan Hacigümüş. 2016. GeoScale: Providing geo-elasticity in distributed clouds. In *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E'16)*.

Keqiang He, Alexis Fisher, Liang Wang, Aaron Gember, Aditya Akella, and Thomas Ristenpart. 2013. Next stop, the cloud: Understanding modern web service deployment in EC2 and azure. In *Proceedings of the Internet Measurement Conference (IMC'13)*.

Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A. Kim. 2012. Measuring interference between live datacenter applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 51.

S. Kundu, R. Rangaswami, K. Dutta, and Ming Zhao. 2010. Application performance modeling in a virtualized environment. In *Proceedings of the Conference on High Performance Computer Architecture (HPCA'10)*.

John D. C. Little. 1961. A proof for the queuing formula: $L = \lambda W$. *Operat. Res.* 9, 3 (1961), 383–387.

Guoxin Liu, Haiying Shen, and Harrison Chandler. 2013. Selective data replication for online social networks with distributed datacenters. In *Proceedings of the 2013 21st IEEE International Conference on Network Protocols (ICNP'13)*. IEEE, 1–10.

Maxmind GeoIP Service 2016. Maximind GeoIP Service. Retrieved from https://www.maxmind.com/en/home.

Memcached 2015. Memcached. Retrieved from http://memcached.org/.

Xiaoqiao Meng, Canturk Isci, Jeffrey Kephart, Li Zhang, Eric Bouillet, and Dimitrios Pendarakis. 2010. Efficient resource provisioning in compute clouds via VM multiplexing. In *Proceedings of the 2015 IEEE International Conference on Autonomic Computing (ICAC'10)*. 10. DOI: http://dx.doi.org/10.1145/1809049.1809052

Ningfang Mi, Giuliano Casale, Ludmila Cherkasova, and Evgenia Smirni. 2008. Burstiness in multi-tier applications: Symptoms, causes, and new models. In *Proceedings of the Conference on Middleware*.

Barzan Mozafari, Carlo Curino, and Samuel Madden. 2013. DBSeer: Resource and performance prediction for building a next generation database cloud. In *Proceedings of teh Conference on Innovative Data Systems Research (CIDR'13)*.

Mohamed N. Bennani and Daniel A. Menasce. 2005. Resource allocation for autonomic data centers using analytic performance models. In *Proceedings of the 2015 IEEE International Conference on Autonomic Computing (ICAC'05)*.

Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds: Managing performance interference effects for QoS-aware clouds. In *Proceedings of EuroSys*.

Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, and Amr El Abbadi. 2015. Minimizing commit latency of transactions in geo-replicated data stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. ACM, New York, NY, 1279–1294. DOI: http://dx.doi.org/10.1145/2723372.2723729

Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. 2013. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the USENIX International Conference on Automated Computing (ICAC'13)*.

P. N. Shankaranarayanan, Ashiwan Sivakumar, Sanjay Rao, and Mohit Tawarmalani. 2014. Performance sensitive replication in geo-distributed cloud datastores. In *Proceedings of the Conference on Dependable Systems and Networks (DSN'14)*.

Percona Xtrabackup. 2015. Percona Xtrabackup (2015). Retrieved from: https://www.percona.com/software/mysql-database/percona-xtrabackup.

Fan Ping, Xiaohu Li, Christopher McConnell, Rohini Vabbalareddy, and Jeong-Hyon Hwang. 2011. Towards optimal data replication across data centers. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems Workshops (ICDCSW'11)*. IEEE, 66–71.

Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. ACM, 85–100.

S. Sakr and A. Liu. 2012. SLA-based and consumer-centric dynamic provisioning for cloud databases. In *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD'12)*.

Ankit Singla, Balakrishnan Chandrasekaran, P. Brighten Godfrey, and Bruce Maggs. 2014. The internet at the speed of light. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets-XIII'14)*. ACM, New York, NY, Article 1, 7 pages. DOI: http://dx.doi.org/10.1145/2670518.2673876

Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM. 385–400. DOI: http://doi.acm.org/10.1145/2043556.2043592

The ObjectWeb TPC-W implementation 2005. The ObjectWeb TPC-W implementation. Retrieved from http://jmob.ow2.org/tpcw.html.

B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. 2005. Dynamic provisioning of multi-tier internet applications. In *Proceedings of the 2015 IEEE International Conference on Autonomic Computing (ICAC'05)*.

Daniel Villela, Prashant Pradhan, and Dan Rubenstein. 2007. Provisioning servers in the application tier for e-commerce systems. *In Proceedings of TOIT* (2007).

Website Latency With and Without a Content Delivery Network 2016. Website Latency With and Without a Content Delivery Network. Retrieved from https://www.keycdn.com/blog/website-latency/.

Timothy Wood, Ludmila Cherkasova, Kivanc Ozonat, and Prashant Shenoy. 2008. Profiling and modeling resource usage of virtualized applications. In *Proceedings of the Conference on Middleware*.

P. Xiong, Y. Chi, S. Zhu, H. J. Moon, C. Pu, and H. Hacigümüş. 2011. Intelligent management of virtualized resources for database systems in cloud environment. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*. 87–98. DOI: http://dx.doi.org/10.1109/ICDE.2011.5767928

Qiang Xu, Jeffrey Erman, Alexandre Gerber, Zhuoqing Mao, Jeffrey Pang, and Shobha Venkataraman. 2011. Identifying diverse usage behaviors of smartphone apps. In *Proceedings of the ACM Internet Measurement Conference (IMC'11)*.

Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. 2007. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Proceedings of the 2015 IEEE International Conference on Autonomic Computing (ICAC'07)*.

Qian Zhu and Teresa Tung. 2012. A performance interference model for managing consolidated workloads in QoS-aware clouds. In *Proceedings of the 2012 IEEE 5th International Conference on Cloud Computing (CLOUD'12)*. IEEE, 170–179.

Received August 2016; revised May 2017; accepted May 2017