2015 IEEE 12th International Conference on Autonomic Computing

Model-driven Geo-Elasticity In Database Clouds

Tian Guo Prashant Shenoy
College of Information and Computer Sciences
University of Massachusetts Amherst
{*tian,shenoy*}*@cs.umass.edu*

***Abstract*—Motivated by the emergence of distributed clouds, we argue for the need for geo-elastic provisioning of application replicas to effectively handle temporal and spatial workload fluctuations seen by such applications. We present DBScale, a system 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 to infer the database query workload from observations of the spatially distributed front-end workload and a two-node open queueing network model to provision databases with both CPU and I/O-intensive query workloads. 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.**

***Keywords*-Distributed Clouds; Database Elasticity**

I. 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 [1] 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 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 eleven and seventeen global locations to their customers today. Distributed clouds are especially well suited for deploying cloud applications that service a geographically diverse workload. For such applications, a distributed cloud platform enables application replicas to be deployed at different cloud locations so that users can be serviced from the nearest cloud replica for the best performance. Studies [2] have shown that such *geo-distributed* application 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 [3] [4], 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). We argue that local cloud elasticity mechanisms that are designed for handling temporal fluctuations in the workload are not well suited for handling spatial fluctuations seen in today's 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.

To address the specific needs of geo-distributed applications, in this paper, we argue for the need to support geo-elasticity mechanisms that can handle both the temporal and spatial variations in the workload. 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. Our paper specifically targets Database-as-a-service (DBaaS) clouds and focuses on designing a geo-elasticity mechanism for DBaaS clouds. (Refer to Fig. 1 for a high level illustration.) We identify four key challenges in designing geo-elasticity for DBaaS clouds. First, since the database tier of the application only sees the traffic from the front-end tier and does not directly see the end-user traffic, inferring the geographic distribution of database workloads and the associated spatial fluctuations is more challenging than for front-end tiers. Second, prior work on dynamic provisioning [5] has often assumed that the database tier is well provisioned and not a bottleneck. As a result, models for provisioning the front-end application tiers often assume that CPU is the bottleneck resource. Such models may not be well-suited for database tiers as database can either be compute-intensive or I/O-intensive, or a mix of the two, depending on the computational and I/O demands of database queries. 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. Our approach provides application the flexibility to choose an appropriate method—either

978-1-4673-6971-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ICAC.2015.46

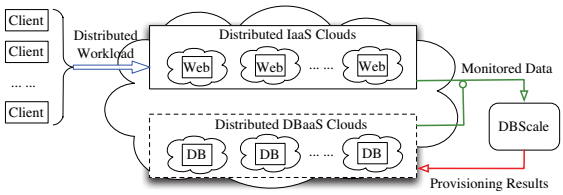


Figure 1. Geographic replication of a multi-tier application's backend tier in distributed clouds using DBScale.

batched updates or more complex configurations such as master-slave or multi-master—for maintaining consistency. Finally, the database tier needs to coordinate its geo-elastic decisions with the front-end to maximize the overall benefits for the multi-tier application.

Contributions. In this paper, we present DBScale, a system for implementing geo-elasticity in a Database-as-a-service cloud. Our system is able to take a *global view* of the application workload to provision an appropriate number of replicas across cloud locations and also to efficiently provision new database replicas in previously unsupported regions when needed. In designing and implementing DBScale, we make the following contributions:

1. 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.

2. 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 node and the disk modeled as a M/G/1-FCFS node. In doing so, our model is able to handle both CPU- and I/O-intensive query workloads seen by database replicas and forms the basis for provisioning capacity within the database tier.

3. We propose our DBScale geo-elastic algorithm for DBaaS cloud using our regression-based workload model and our queueing-based resource model for database replicas. We implement a prototype of DBScale on Amazon EC2's distributed clouds to track application workloads, predict spatial fluctuations, use empirical measurements in conjunction with our models to compute capacity needed across cloud locations, and replicate, terminate or configure databases based on the provisioning results using public cloud APIs.

4. We conduct detailed experimental evaluations of DBScale using the Amazon EC2's global cloud platform. We compare the effectiveness of the geo-elasticity approach to using only local elasticity approach and also compare the use of geo-elasticity approach to a centralized approach that uses a distributed cache. We also demonstrate the efficacy of coordinating DBScale's backend provisioning decisions with those of the front-end tiers. Our results show a 55.03% improvement in mean response time when compared to local elasticity, a 66.22% benefit due to coordination, and a 35.8% benefit when compared to a caching-based approach.

II. 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.

Distributed Database Clouds. Our work assumes a Database-as-a-service cloud that allows application providers, also referred to as *tenants*, to lease one or more databases from the cloud platform. The DBaaS cloud provides SLAs on the performance (e.g., response times) seen by the application and handles the task of configuring and provisioning sufficient capacity for each tenant. Just as 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, an application may choose a particular cloud location that is best suited to its needs or a set of locations where application replicas are placed.

Application Model. Our work focuses on 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. For simplicity, we assume that application uses a single cloud provider that provides IaaS and DBaaS services. This assumption allows the front-tiers and the backend tier replicas to be hosted in the same data center of that cloud provider at each location where the application has a presence. We assume that the multi-tier application has users that are spread across multiple geographic locations and hence it services a geographically diverse workload. Further, in addition to temporal variations such as time-of-day effects, such a geographically diverse workload is assumed to exhibit 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). We assume that the database tier (and the multi-tier application itself) is geo-distributed, with replicas in different regions, to handle the geographically diverse workload. Since the database tier is replicated, both within a particular cloud location and across locations, maintaining consistency of the backend replicas is an important issue. We assume that the consistency policies and mechanisms implemented by the 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.

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 implement elasticity mechanisms where the number of database replicas of the tenant application is scaled up or down in response to workload dynamics. 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* the workload increase or decrease occurs and decide how many replicas are needed at each cloud location where the application has been deployed. If the application sees traffic from new geographic locations, the cloud platform will need to provision replicas in new locations. Such elastic provisioning of capacity within and across cloud locations to handle both temporal and spatial workload fluctuations is referred to as *geo-elasticity*. Provisioning of additional replicas, whether within an existing data center or at new data center, must be efficient since it involves substantial copying of disk state. Further, such provisioning may need to be coordinated with elasticity mechanisms at the front-end tier since it is advantageous to also provision a front-end replica at a new site when provisioning a backend database replica.

Formally, given a DBaaS cloud with N locations, let $\langle r_1, r_2, \ldots r_n \rangle$ denote the front-end workload from users in each of those N geographic regions. Then given this observed front-end workload, we must first infer the spatial distribution of the database query workload seen by the backend tier as $\langle \lambda_1, \lambda_2, \ldots \lambda_n \rangle$ where λ_i database queries are generated by the r_i front-end requests at location i. Given this query workload distribution, the goal of the geo-elasticity algorithm is to determine the number of database replicas to provision at each location $\langle d_1, d_2, \ldots d_n \rangle$, where d_i denotes the number of database replicas at location i. If $d_i \neq d'_i$ where d'_i denotes the current number of database replicas at location i, then $|d_i - d'_i|$ additional replicas need to be provisioned (or deactivated) at location i. If $d'_i = 0$, then location i is a new location for the application (i.e., application does not have any replicas at this site presently) , and d_i new database replicas need to be placed at this geographic location.

DBScale Architecture. Fig. 2 shows the design of DBScale. It dynamically provisions databases in cloud locations that see temporal and spatial variations of user workload. We explain in details about monitoring and predicting geo-dynamics database workload in Section III, queueing-based model that handles CPU-intensive and I/O-intensive workloads in Section IV and step-by-step procedures to provide geo-elastic database clouds in Section V.

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

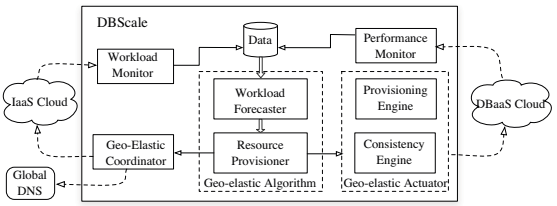


Figure 2. Key components of the DBScale architecture.

III. Regression-based prediction for Geo-dynamic Database workload

To obtain the temporal and spatial database workload, it is usually not enough to just monitor the backend database workload. This is because queries that are sent to the backend do not contain the end-users' IP addresses. In this section, we introduce our regression model that enables DBScale to predict both the *temporal and spatial* workload changes that will be seen by the database tenants by observing the front-end workload.

Assume that there are N cloud locations and the application presently has replicas at k sites. We first aggregate the front-end request logs from all k locations; the request logs are assumed to include the IP address of the end clients that issued the request. The aggregated log, which represents the global workload of the application, is then processed by IP Geolocation techniques to determine the geographic location of each request.[2] The geographic location of each request is then mapped to the nearest cloud location, irrespective of whether the application has an active replica at that site. This process effectively partitions the global workload of the application into N bins and yields a request rate vector $\langle r_1, r_2, \ldots, r_N \rangle$ that represents the workload spatial distribution across various clouds. Given the front-end workloads, we wish to infer the database query rates $\langle \lambda_1, \lambda_2, \ldots \lambda_N \rangle$ that would be seen at those N locations.

Since each front-end request triggers *one or more* database queries, hence the front-end requests and the backend queries seen within each observation interval T are correlated and we attempt to learn these correlations using regression techniques. We use Equation 1 to capture the relationship between request rate r_i seen at location i and the query rate λ_i that it triggers.

$$\lambda_i = a_i r_i + b_i, \quad i = 1, 2 \ldots N \tag{1}$$

where the term a_i captures the linear relationship and b_i is an error term. Let us define a binary indicator variable X_{ij}:

$$X_{ij} = \begin{cases} 1 & \text{if requests from } i^{th} \text{ region is served} \\ & \text{by } j^{th} \text{ cloud location.} \\ 0 & \text{Otherwise.} \end{cases}$$

[2]IP Geolocation is a technique that infers user's geographic location from IP address. We use MaxMind GeoIP2 [6] for this task.

We can then express the actual query rate seen by the back-end tier at the active cloud location j, λ'_j, in terms of the unknown query rate λ_i as in Equation 2.

$$\lambda'_j = \sum_{i=1}^{N} X_{ij}\lambda_i \quad j = 1, 2, \ldots k \tag{2}$$

Combining Equation 1 and Equation 2, we obtain the multiple linear regression model in Equation 3. (The regression model is illustrated in Fig. 3a with known parameters marked in green and unknown in red.)

$$\begin{aligned}\lambda'_j &= \sum_{i=1}^{N} X_{ij}(a_i r_i + b_i) \\ &= \sum_{i=1}^{N} a_i r_{ij} + \sum_{i=1}^{N} b_{ij}\end{aligned} \tag{3}$$

where r_{ij} and b_{ij} denote the amount of front-end workload from region i that is served in cloud location j as well as the corresponding error term. For each active cloud location j, we can then use Least Squares Regression[3] to obtain corresponding coefficients $\hat{a}_i$ (that satisfies $X_{ij} = 1$) and the collective error term $\hat{b}'_j$ using data from each measurement interval T that minimize the error ϵ as in Equation 4.

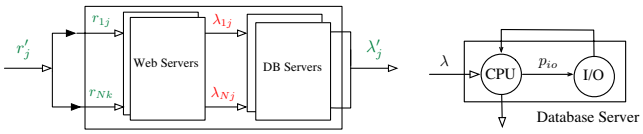
$$\epsilon = \sqrt{\sum_{t=1}^{T} \Big(\sum_{i=1}^{N} \hat{a}_i r_{ijt} + \hat{b}'_j - \lambda'_{jt} \Big)^2} \tag{4}$$

To approximate $\hat{b}_i$, we use $w_i < 1$ to denote its fraction in the collective error term $\hat{b}'_j$, e.g. we could set $w_i = \frac{1}{n}$ if requests from n locations are observed in location j, and assign $\hat{b}_i = w_i \hat{b}'_j$. The resulting regression model $(\hat{a}_i, \hat{b}_i)$ for i^{th} location then allows us to simply take the spatial workload seen by the front-end and use the model to estimate the geographic query workload that will be seen by the backend tier. The future workload can be predicted using the time series of the front-end workload emanating from a location i as $\langle r_i(t-M), r_i(t-M+1), \ldots r_i(t)\rangle$, where M represents the historical time window, and using any standard time series prediction technique, such as ARIMA models [7] to predict the workload in the next interval as $\hat{r}_i(t+1)$ and the regression model can be used to estimate the future backend workload $\hat{\lambda}_i(t+1)$ from each location. The predicted query workload is then used as a basis to provision database capacity in a geo-elastic manner.

IV. Queueing-based Database Provisioning

Given the predictions of the query workload at various locations, the DBaaS cloud employs a queueing-based capacity model to determine the number of replicas to provision to handle the incoming workload. Queueing-based models have been used previously for dynamic resource provisioning [5] [8] [9], but these are designed for provisioning front-end tier where CPU is assumed to be the bottleneck resource. In contrast, the database tier may see CPU-intensive or I/O-intensive queries, or a mix of the two, and may need models to consider the impact of both resources; prior modeling approaches that mostly focus on the CPU are not well suited for this scenario. We present a database-specific queueing-based model that can handle both CPU-intensive and I/O-intensive workloads.

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



(a) Regression-based model for j^{th} cloud site. (b) Single database server Queueing model.

Figure 3. **Model-driven geo-elastic approaches**. We perform regression-based prediction for each database cloud location to obtain the peak database workload. For estimating the per-server capacity at location j, we model the server as a two-node open queueing network.

We model the database tenant replica (on a dedicated host) as a two-node open queueing network with feedback, where the CPU is modeled as a M/G/1-PS node and I/O device as a M/G/1-FCFS node as in Fig. 3b. The queueing network captures the processing of a query which involves disk I/O to read data from database and CPU processing to process the data and execute the query. We use a recent result from the queueing literature [10] to derive an approximation of the mean sojourn time spent by the query on the CPU and the disk for such a two-node queueing network. Let us denote $E[T_{cpu}]$ and $E[T_{io}]$ as the mean sojourn CPU and I/O time, respectively. Then the result states that:

$$\begin{aligned}E[T_{cpu}] &= \frac{\bar{s}_{cpu}}{(1-p_{io})(1-\rho_{cpu})} \\ E[T_{io}] &= \frac{p_{io}}{1-p_{io}}\Big[\frac{\bar{s}_{io}}{1-\rho_{io}} + \frac{p(\bar{s}^{(2)}_{io} - 2\bar{s}^2_{io})}{2(1-\rho_{io})}\lambda\Big]\end{aligned} \tag{5}$$

where $\bar{s}_{cpu}$ and $\bar{s}_{io}$ denote the average service time of CPU and I/O queues; ρ_{cpu} and ρ_{io} denote the average utilization of CPU and I/O queues; $\bar{s}^{(2)}_{io}$ is the second moment of the service time distribution of I/O queue and p_{io} is the query visit ratio to the I/O.

The mean sojourn time $E[T]$ of database queries is then the sum of time spent in CPU and I/O queues, i.e., $E[T] = E[T_{cpu}] + E[T_{io}]$. Since the total time spent by the query in the database tier has to satisfy the SLA y, which we define to be the 95^{th} percentile response time, $E[T]$ needs to satisfy the constraint $\alpha_T(95) < y$. For the 95^{th} percentile, we have $E[T] < \frac{y}{3}$. [4] Hence $E[T_{cpu}] + E[T_{io}] < \frac{y}{3}$, and Equation 6 yields an upper bound on the maximum query rate λ^c_i that can be handled by a single database replica at location i without violating the SLA:

$$\lambda^c_i \leq \frac{\frac{2y}{3}(1-p)(1-\rho_{io}) - 2\bar{s}_{cpu}\frac{1-\rho_{io}}{1-\rho_{cpu}} - 2p\bar{s}_{cpu}}{p^2(\bar{s}^{(2)}_{io} - 2\bar{s}^2_{io})} \tag{6}$$

Obtaining Model Parameters. In order to acquire all the model parameters for estimating $E[T]$, we need to

[4]For a general distribution, we can utilize Markov Inequality to obtain $\alpha_T(95) \leq 20E[T]$. Here we assume an exponential distribution for response times, which yields a tighter bound.

empirically measure the CPU utilization, I/O utilization (using Linux tools such as sysstat) as well as the per-query log that includes the query timestamp 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 the CPU utilization log 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. p_{io} is estimated as $\frac{\lambda_{io}}{\lambda+\lambda_{io}}$. Since we do not have easy access to $\bar{s}_{cpu}$ and ρ_{cpu}, we approximate these two parameters by using little's law as $\bar{s}_{cpu} = \frac{\rho_{cpu}}{\lambda+\lambda_{io}}$ and $\rho_{io} = \lambda_{io}\bar{s}_{io}$. It is important to note that the measurement data are an overestimate due to the extra logging overhead as well as the resource interference. Finally, to model the database replica for a new datacenter location, we use the average of the measured statistics across all available data centers as an initial approximation.

V. Geo-elastic Provisioning Algorithm

In this section, we describe how DBScale utilizes the workload monitoring and predictions and the queueing-based model to implement the geo-elastic provisioning algorithm. The provisioning algorithm is invoked periodically or when SLA violations are observed.

Step 1: Where to provision? As discussed earlier, DBScale tracks the query workload emanating from a geographic region associated with each of the N cloud locations. Based on these measurements and future predictions, we decide where (i.e., which subset of these N locations) to place replicas. The decision of whether to place a replica at a cloud site will depend on whether there is sufficient user traffic from that region. We sort the cloud locations in increasing order of workload volume and use a simple threshold-based approach to make this decision—if the query workload volume at a location i is less than a low threshold τ, then no replica is placed at that location. Instead, workload from that region is reassigned to the next closest cloud location and the process is repeated. At the end of this process, we are left with a subset of regions that have sufficient workload to justify placing one or more replicas at each site.

Step 2: How much to provision for each location? We recompute the workload vector $\hat{\lambda}$ obtained from Sec III so that $\hat{\lambda}_i$ comprises the workload from its geographic region and the workload that has been mapped to this location from other regions as a result of the above step. If a decision was made to not place a replica at a location i, $\hat{\lambda}_i$ is set to 0. Next we use the empirical measurements of workload $\hat{\lambda}_i$ to estimate parameters for our queueing model and use the model to compute the maximum query rate λ_i^c that can be handled by a single replica while meeting the SLA y. Given this capacity of a single database replica, the number of replicas d_i needed at this location is computed as:

$$d_i = \lceil \frac{\hat{\lambda}_i}{\lambda_i^c} \rceil \tag{7}$$

If the number of replicas d_i differs from the value d_i' computed in the previous time interval (i.e. current provisioning), $|d_i - d_i'|$ more replicas need to be provisioned or deactivated at this location. If $d_i' = 0$, this indicates that location i has been newly chosen to provision database replicas.

Step 3: Coordinating with front-end tier. While the DBaaS cloud can make independent provisioning decisions, coordinating with the front-end tier by informing it of the decisions that have been made is desirable. For example, if the DBaaS cloud decided to place a replica at a newly chosen location, the front-end tier must be informed so that it can provision a front-end replica at that site to take advantage of the new database replica. Similarly, the front-end tier may decide to geo-elastically place a replica at a new cloud location that previously lacked application presence, and it needs to inform the backend tier to "force" the provisioning of a database replica at that site regardless of the backend provisioning decisions.

Step 4: How to provision database replicas? To provision a new replica, we first make a hot backup from an existing replica using a hot backup tool. (e.g., we use XtraBackup[5] for MySQL database) The hot backup tool produces a consistent point-in-time snapshot of the database without interrupting normal database processing at that replica. Then the snapshot is transferred 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. The snapshot is loaded into the database by using the hot backup tool's crash recovery feature. If any updates are made to the database replica in the meantime, DBScale use an offline approach that acquire a read-lock on current replicas, fetch write queries and apply 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).

We note that the provisioning latency of backend servers might be hours if the data size to be copied is large. To alleviate the problem, we propose to archive one copy of up-to-date snapshot in every data center location by periodically transferring new snapshots from the running database servers. And at provisioning time, we only need to copy the difference between the new snapshot and archived snapshot and use it as a basis to start the database in that location. By using this optimization, we can better synchronize the provisioning time of front-end and back-end servers and quickly use the new servers to service workload increases.

VI. DBScale Implementation

We have implemented a prototype of DBScale on Amazon EC2's distributed clouds. We assume that our DBaaS cloud is implemented using IaaS servers where database replicas run on servers leased from the IaaS cloud. Our prototype is based on the MySQL database platform—that is, database tenants are provided as MySQL databases by the DBaaS cloud. Our system is implemented in Python and is depicted in Fig. 2. Resource usage statistics at the database servers

[5]http://www.percona.com/doc/percona-xtrabackup/2.2/

hosting the tenant replicas are measured using *sar* and *iostat* utilities, which yield the database server's CPU and I/O utilization. DBScale's workload monitoring involves gathering workload statistics from the front-end and backend tiers of each application tenant. The workload monitoring component gathers web server logs from front-end replicas at each location and aggregates them, as discussed in Section III, for analyzing the geographic distribution of the workload. It also gathers database query logs from backend replicas, which contains informations about each query executed, including the execution and start time. A system daemon collects data from various replicas and cloud locations and transfers them to a central controller, which then stores them in a SQLite database. These statistics are then used to construct a regression model as well as a time-series predictor using Python's StatsModels library; those models are then utilized to produce the future peak temporal and spatial database workload for geo-elastic provisioning. The provisioning engine implements our queueing-based model in Python and implements the provisioning algorithm from Section V to compute the maximum capacity that can be sustained by a database replica and the number of replicas needed at each site. The replicas are provisioned using Amazon's EC2 command line utilities to start up servers and database hot backup tools to extract snapshots and load them into new replicas. Maintaining the consistency of database replicas is a key task. Our system supports a batched mode that can be used during maintenance windows; in this case, updates are made as a batch at replicas in an offline mode during maintenance downtime. Alternatively, replicas can be configured in a MySQL's master-slave configuration and updates are made to the master and relayed to the slaves. In this online mode, DBScale picks a database from a geographical central location and configures it as master database; the remaining replicas at this and other locations become slaves. The choice of a specific method depends on the application's needs. For example, when used for holding data such as product catalogs that see largely read queries, a simple batched update approach may suffice.

VII. Experimental Evaluation

Experiment Setup. We experimentally evaluate DBScale on Amazon EC2's distributed cloud. We use TPC-W [11] as our multi-tier application; the front-end tier of TPC-W runs on Apache Tomcat and is hosted on Amazon's IaaS cloud. The backend tier of TPC-W is hosted using a DBaaS cloud that we create on Amazon cloud using MySQL database servers. DBScale manages the replicas of each tenant in this DBaaS cloud. Both the front-end and backend tiers of TPC-W are assumed to be geo-elastic and replicable both within and across EC2 cloud locations as needed. To inject geo-dispersed client workload, we run the TPC-W clients on PlanetLab's servers that spread across multiple locations. Client requests from a PlanetLab node are forwarded to the nearest front-end replica using a custom DNS-based load redirection. In our experiments, we use default browsing and ordering workload mix for TPC-W as well as a modified read-only browsing workload. In the rest of this section, we use this experimental setup to evaluate the effectiveness of our regression and queueing models, the overall effectiveness of our geo-elasticity approaches under various scenarios, provide a comparison with a caching-based centralized approach and measure consistency overhead due to our geo-elastic replication.

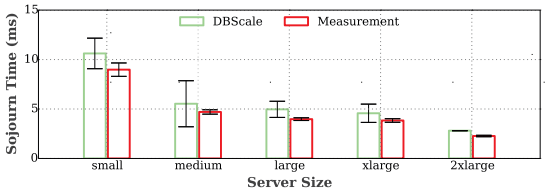


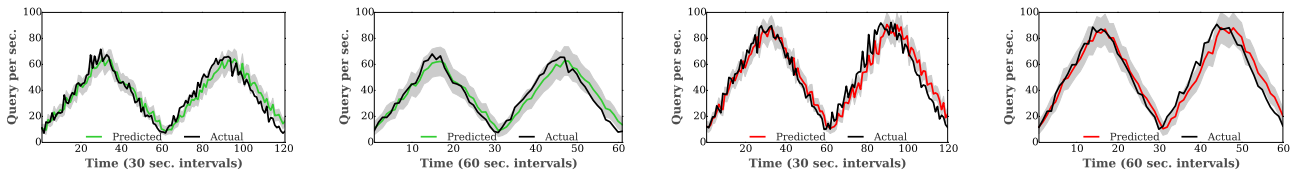
Figure 5. **Efficacy of the queueing model.** For each database server size, we compare the empirically measured response times with queueing model predictions.

A. *Effectiveness of Our Models*

We first study the effectiveness of our regression-based workload model and queueing-based capacity model. To conduct our experiment, we set up multiple PlanetLab client nodes to inject workload to TPC-W application with the front-end web and back-end database servers hosted in the EC2 Virginia data center. We vary the workload intensity by adjusting the number of concurrent clients that connect to the front-end tier from 10 to 100 in increments of 10. We collect the necessary logs for training and testing our regression-based workload model as well as queueing-based capacity model.

Effectiveness of the Regression Model. We conduct each experiment run for one hour and repeat it five times each for different server sizes e.g. small or medium size servers. Data from the first four runs are used to train the regression model and obtain the model parameters a and b for this cloud location,as illustrated in Fig. 3a. We then use the trained model to predict the database query rate for the final run and compare the predictions to the empirically measured database query rates. Fig. 4 shows that our regression model makes good predictions for different workload mixes and both prediction intervals. Overall, the regression model predictions have a mean error of 7.35%.

Efficacy of the Queuing Model. We next use the queueing model that takes the above prediction to compute the response time (which in turn yields server capacity) on database servers of different size. For each server size, we run five 30-minutes experiments and calculate the mean response time using Equation 5. In Fig. 5, we compare the average approximation values across five runs to the empirically measured value. We see that empirically measured response times lie within the 95% confidence interval values of the model predictions, 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.



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

Figure 4. **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 95^{th} percentile confidence interval. For both workload types, the prediction accuracy is higher for a larger prediction window.

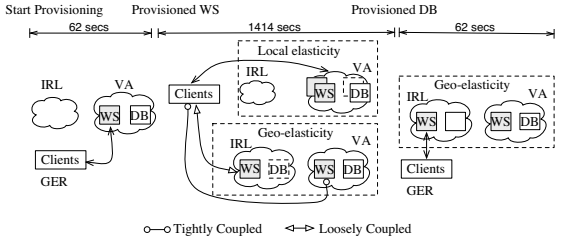


Figure 6. **Elasticity mechanisms and provisioning policies.** We use 10 clients from Germany to inject traffics. It takes 62 seconds to provision web server and an extra 1414 seconds to provision database server with 10GB data. Web servers are connected to different databases based on the provisioning policy.

	Mean (ms)	Std. Dev. (ms)	95% Conf. Int. (ms)
Local Elasticity	169	20	[167.07, 170.93]
Geo-elasticity	76	10	[75.34 , 76.66]

Table I
Comparisons of client response times for different elasticity approaches. Geo-elasticity provides lower mean response times due to lower client-server network latencies.

B. *Benefits of Database Geo-elasticity*

We next conduct experiments to compare local and geo-elasticity mechanisms and study the performance differences between loosely and tightly coupled provisioning policies.

Database Geo-elasticity Benefits. Our first experiment compares local elasticity and geo-elasticity techniques. We provision the application in the Virginia, USA data center and inject a light workload from the US. We then start 10 PlanetLab client nodes in Germany, which causes the application to see traffic from Europe. Both local and geo-elasticity mechanisms can react to this workload increase by provisioning additional capacity, as shown in Fig. 6. In case of local elasticity, the additional capacity is provisioned locally (within the same Virginia data center location), while geo-elasticity techniques can provision capacity at any suitable cloud location. As a result, local elasticity technique will provision additional database replica (and a front-end node) at Virginia, while our approach does so at EC2's Ireland data center. Once this is done, requests from the German PlanetLab nodes are routed to the Ireland front-end and database replicas. Table I shows that the mean client response time is improved by 55.03% when switching from local elasticity to geo-elasticity provisioning. This is mainly due to the 70 ms network RTT reduction, from 100.29 ms to 29.67 ms due to serving European request and query traffic from an European cloud location in Ireland.

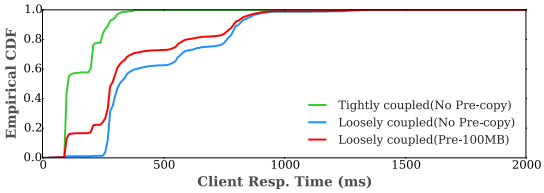
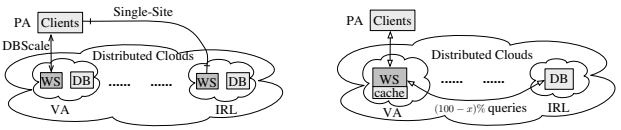


Figure 7. **CDF comparison of client response times for loosely and tightly coupled provisioning policies.** A tightly coupled policy improves the 95^{th} percentile of response time from 1350 ms to 390 ms when compared to the loosely coupled policy. Pre-copying improves performance of the loosely coupled policy.

Loosely v.s. Tightly Coupled Provisioning Policy. Next, we study loosely and tightly coupled provisioning policies. In the loosely coupled approach, the front-end and backend tier make *independent* provisioning decisions, while in the tightly coupled approach, they coordinate the provisioning of replicas at both tiers. In our case, provisioning a front-end node takes only 62s, while provisioning a 10GB database in the backend tier takes an extra 1414 seconds. (Refer to Fig. 6.) We repeat the same scenario as before where as start with the application in the Virginia data center and inject additional traffic from Europe using German PlanetLab nodes. In the loosely coupled approach, both tiers react to this workload and initiate provisioning of a replica in Europe. However, the front-end replica comes up first and since the database replica is not ready, it will need to send its queries to the backend hosted in Virginia Cloud in USA, incurring a large WAN latency between the front-end and backend nodes. In the tightly coupled approach, both tiers wait until replicas have been provisioned before activating and redirecting user traffic to the Ireland cloud.

Fig. 7 shows that the tightly coupled policy performs better than the loosely coupled policy with an 71.11% improvement of 95^{th} percentile response time. Moreover, most client requests incur more than 260 ms latency in the loosely coupled policy. This is caused by a total of 140 ms network RTT, dominated by the distance between web and database server, between end-users and the front-end server, and the fact that single http request might trigger multiple database queries. Since the tightly-coupled coordinates provisioning across tiers, the switchover to the new replica takes longer, but the new replica yields good performance as expected. We next conduct an experiment to study the potential benefits



(a) Baseline setups: DBScale and single-site database elasticity.

(b) Set up for $x\%$ cache hit rate.

Figure 8. **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.

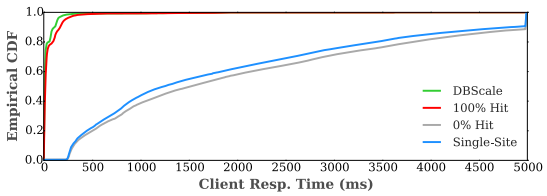


Figure 9. **CDF comparison of end-user response time of four different scenarios.** A caching approach with 100% hit rate has comparable performance to DBScale while a 0% hit rate causes performance to be similar to local single-site elasticity.

of pre-copying database snapshots to the destination cloud location for both policies. As shown in Fig. 7, since we only need to copy a delta of 100 MB data to provision the database in the new location, the CDF of client response time for loosely coupled policy is improved in the presence of pre-copying. While pre-copying databases does not impact the response time distribution in the tightly coupled policy, it does speed up the provisioning of database replicas and thus reduce the latency to activate the front-end and backend replicas at the new location.

C. Comparing DBScale to a Caching Approach

In this experiment, we compare DBScale's geo-elastic replication to a caching-based approach. In a caching approach, we assume that the backend tier is *centralized* and present only at a single location. The front-end tiers, on the other hand, are replicated in various geographic locations. Since the WAN latency between the front-end and backend tiers is high, each front-end tier is assumed to employ an in-memory cache, such as Memcached [12], where recent query results are stored. In this case, upon receiving a request, the front-end tier first examines the cache and only issues a query to the remote backend tier when data is not cached. By centralizing the backend database, the approach does not suffer from consistency problems and when hit rates are high, the geo-replicated front-end tier can service users from nearby cloud locations. We modified the TPC-W application so that it always check the in-memory cache first before accessing the database in the remote cloud location. We use a small database of 512 MB and allocate 1 GB RAM for our in-memory cache. We use the browsing workload mix and pre-profile all the queries and their parameters. We can control the cache hit rate by pre-loading a subset of the query results into the cache and controlling the request mix so that a certain hit rate is achieved. We compare the caching approach to our geo-elastic replicated approach (see

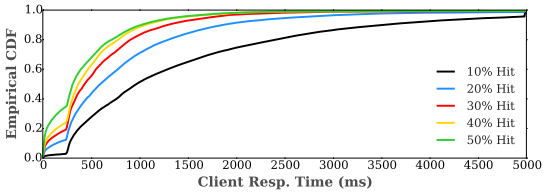
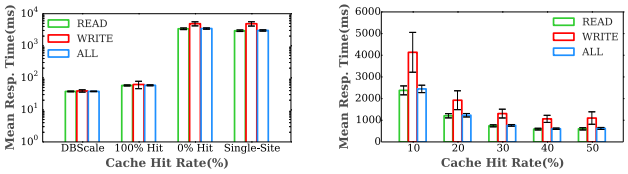


Figure 10. **CDF comparison of end-user response time with increasing hit rate.** As the hit rate increases from 10% to 50%, the 95^{th} percentile response time improves by 72.18%, from 4780 ms to 1330 ms.



(a) Caching and baselines scenarios.

(b) Various cache hit rates.

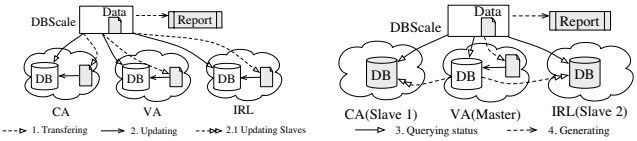
Figure 11. **Comparisons of mean end-user response time.** We calculate the mean response times for read and write requests. In accordance with the results from Fig. 9 and Fig. 10, DBScale has the lowest mean response time for all request types. Interestingly, the response times for write requests improves but flattens out around 1065.4 ms. This is due to a lighter workload at remote databases together with the inherent network latency.

Fig. 8b) and also provide baseline results for a local elasticity approach where both front-end and backend tier are housed at a single site.

In-Memory Cache v.s. DBScale. Fig. 9 provides a CDF of the end-user response times for the caching-based approach with DBScale and local elasticity approaches. For the caching approach, we show the results for two extreme workloads that achieve cache hit rates of 0% and 100%. A 0% hit rate sees poor response times since the queries sent by the front-end tier to the remote database see large WAN latencies. The performance is similar to single-site elasticity where clients' requests incur a similar WAN latency to a far-away application. In contrast, when the hit rate is 100%, the performance of the caching approach is comparable to the DBScale geo-replicated application. In both cases, end-client requests go to a nearby front-end replica and the latency to obtain query results from a local cache or a local database are both small.

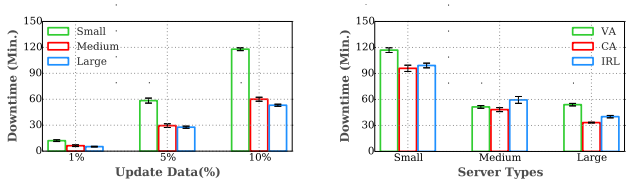
Impact of Cache Hit Rate. We next inject workloads that see different cache hit rates for the caching-based approach. As the hit rate increases from 10% to 50%, the client response time also improves accordingly as shown in Fig. 10. This is because the percentage of requests that avoid a WAN hop to the remote database decreases as the hit rate increases from 10% to 50%. Importantly, the benefits of caching only accrue for predominantly read-intensive query workloads. Queries that make updates will always need to be sent to the remote database and incur poor response times as shown in Fig. 11b.

Overall a caching based approach can provide comparable performance to DBScale's geo-elastic replication when cache hit rates are high. Performance of the caching approach suffers significantly for lower cache hit rates or when a higher fraction of write queries are present in the workload.



(a) Batched updates setup. (b) Online master-slave setup.

Figure 12. **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.



(a) Maintenance down time. (b) Update 10% of database.

Figure 13. **Batched updates Overheads.** The maintenance downtime due to batched updates is impacted by the server capacity, i.e. server size and the cloud location, and the amount of the data that need to be updated.

DBScale's geo-replication does not depend on these factors and yields good performance always (at the cost of needing consistency maintenance among replicas). Further, we used an artificially small database in this experiment to understand the best case scenario for the caching approach. In practice, database sizes will be much larger than the size of the in-memory cache, and the actual cache hit rates will depend on the skew in the query popularity distribution (and the size of the in-memory cache).

D. 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, i.e. batched and online updates using MySQL master-slave configuration, to achieve database consistency. In both experiments (as shown in Fig. 12), we use TPC-W web application benchmark that is loaded with 13.43GB database.

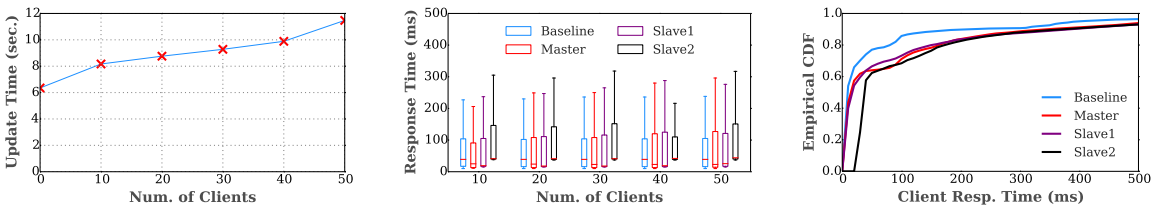
Batched Updates Overhead. We measure the overhead of applying a varying amount of updates in batch mode during offline maintenance windows; the three database replicas are each hosted separately in Virginia, California and Ireland data centers, as shown in Fig. 12a. We measure the latency to apply updates at all replicas and restart all servers. Fig. 13a 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 12.52 minutes to update 1% of database data on a small server and as much as 60 minutes 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 Fig. 13b. These results show that, barring undersized small severs, batched updates can be a feasible option during maintenance windows, which themselves last for a few hours.

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 Fig. 12b, 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. 14a shows that it takes 6.35 minutes to update 1% data and as the front-end workload increase, the online update time increase too. Our observation suggests that in order to reduce the length of update time, i.e. 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 Fig. 14b, 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 Fig. 14c, we show that the 95^{th} percentile response time increases from 400 ms to 560 ms for master and to 595 ms for slaves for a 50 clients workload at each location.

VIII. Related Work

Distributed cloud platforms have became a popular paradigm for hosting web applications with dynamic workloads [3], [4]. In virtualized cloud, one challenging problem is to accurately model the resource usages of each VM [13]–[16]. The problem becomes more noticeable for applications with bursty workload characteristics [17]. To overcome this hurdle, recent efforts have attempted to mitigate the impact of interference either by combining the VMs workloads [18] or by employing a novel performance prediction model [19]. In our work, we combine measured distributed front-end workload and a regression-based model to predict the spatial and temporal variations for the backend database workload. Queueing-based models have been used extensively to model cloud-based applications [5], [8], [9], but most have focused on front-end servers . An alternative regression model [20] was proposed to approximate the CPU demand of complex systems with different transaction mixes. 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 are offloaded to the cloud, researchers have begun to focus on adaptive and dynamic provisioning of database servers based on SLA [21]–[24]. These efforts on database provisioning include using models and tools to predict resource utilization



(a) Online maintenance Time. (b) Impact on client response time. (c) Client response time CDF.

Figure 14. **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.

and performance for databases [21], [25], cloning techniques to spawn database replicas [23], live migration techniques to horizontally scale up database server [26], middleware approach to coordinate cloud-hosted applications and databases without violating SLA [22] and utilizing distributed cloud platforms for performance-aware data replication [24]. 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.

IX. Conclusions

Motivated by the emergence of distributed clouds, this paper focused on implementing geo-elasticity for geo-distributed multi-tier applications that host their backend tier in DBaaS clouds. We proposed a regression-based model to infer the database query workload from observations of the spatially distributed front-end workload and a two-node open queueing network model to provision databases with both CPU and I/O-intensive query workloads. We implemented a prototype of our DBScale geo-elasticity system and conducted experimental evaluations using Amazon EC2 servers and PlanetLab clients across different continents. 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 study integrated approaches for geo-elastic provisioning of different tiers of a multi-tier application.

Acknowledgment

We thank our reviewers for their comments. This work is supported by NFS grant #1345300, #1229059 and #1422245.

References

[1] K. He, A. Fisher, L. Wang, A. Gember, A. Akella, and T. Ristenpart, "Next stop, the cloud: Understanding modern web service deployment in ec2 and azure," in *IMC*, 2013.
[2] Q. Xu, J. Erman, A. Gerber, Z. Mao, J. Pang, and S. Venkataraman, "Identifying diverse usage behaviors of smartphone apps," in *IMC*, 2011.
[3] M. F. Arlitt and C. L. Williamson, "Internet web servers: Workload characterization and performance implications," *IEEE/ACM Trans. Netw.*, 1997.
[4] R. Birke, L. Y. Chen, and E. Smirni, "Usage patterns in multi-tenant data centers: A temporal perspective," in *ICAC*, 2012.
[5] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal, "Dynamic provisioning of multi-tier internet applications," in *ICAC*, 2005.
[6] "Maximind geoip service," https://www.maxmind.com/en/home.
[7] G. E. P. Box and G. Jenkins, *Time Series Analysis, Forecasting and Control*, 1990.
[8] D. Villela, P. Pradhan, and D. Rubenstein, "Provisioning servers in the application tier for e-commerce systems," *TOIT*, 2007.
[9] M. N. Bennani and D. A. Menasce, "Resource allocation for autonomic data centers using analytic performance models," in *ICAC*, 2005.
[10] O. J. Boxma, R. D. van der Mei, J. A. Resing, and K. M. C. van Wingerden, "Sojourn time approximations in a two-node queueing network," in *ITC*, 2005.
[11] "The objectweb tpc-w implementation," http://jmob.ow2.org/tpcw.html.
[12] "Memcached," http://memcached.org/.
[13] S. Kundu, R. Rangaswami, K. Dutta, and M. Zhao, "Application performance modeling in a virtualized environment," in *HPCA*, 2010.
[14] T. Wood, L. Cherkasova, K. Ozonat, and P. Shenoy, "Profiling and modeling resource usage of virtualized applications," in *Middleware*, 2008.
[15] L. Cherkasova and R. Gardner, "Measuring cpu overhead for i/o processing in the xen virtual machine monitor," in *ATEC*, 2005.
[16] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for qos-aware clouds," in *EuroSys*, 2010.
[17] N. Mi, G. Casale, L. Cherkasova, and E. Smirni, "Burstiness in multi-tier applications: Symptoms, causes, and new models," in *Middleware*, 2008.
[18] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis, "Efficient resource provisioning in compute clouds via vm multiplexing," in *ICAC*, 2010.
[19] G. Casale, N. Mi, L. Cherkasova, and E. Smirni, "Dealing with burstiness in multi-tier applications: Models and their parameterization," in *TSE*, 2012.
[20] Q. Zhang, L. Cherkasova, and E. Smirni, "A regression-based analytic model for dynamic resource provisioning of multi-tier applications," in *ICAC*, 2007.
[21] B. Mozafari, C. Curino, and S. Madden, "Dbseer: Resource and performance prediction for building a next generation database cloud." in *CIDR*, 2013.
[22] S. Sakr and A. Liu, "Sla-based and consumer-centric dynamic provisioning for cloud databases," in *CLOUD*, 2012.
[23] E. Cecchet, R. Singh, U. Sharma, and P. Shenoy, "Dolly: Virtualization-driven database provisioning for the cloud," in *VEE*, 2011.
[24] S. P N, A. Sivakumar, S. Rao, and M. Tawarmalani, "Performance sensitive replication in geo-distributed cloud datastores," in *DSN*, 2014.
[25] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan, "Workload-aware database monitoring and consolidation," in *SIGMOD*, 2011.
[26] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi, "Zephyr: Live migration in shared nothing databases for elastic cloud platforms," in *SIGMOD*, 2011.