STeP: Scalable Tenant Placement for Managing Database-as-a-Service Deployments

Rebecca Taft
MIT CSAIL
rytaft@csail.mit.edu

Willis Lang
Microsoft Gray Systems Lab
wilang@microsoft.com

Jennie Duggan
Northwestern University
jennie.duggan@northwestern.edu

Aaron J. Elmore
University of Chicago
aelmore@cs.uchicago.edu

Michael Stonebraker
MIT CSAIL
stonebraker@csail.mit.edu

David DeWitt
Microsoft Gray Systems Lab
dewitt@microsoft.com

Abstract

Public cloud providers with Database-as-a-Service offerings must efficiently allocate computing resources to each of their customers. An effective assignment of tenants both reduces the number of physical servers in use and meets customer expectations at a price point that is competitive in the cloud market. For public cloud vendors like Microsoft and Amazon, this means packing millions of users' databases onto hundreds or thousands of servers.

This paper studies tenant placement by examining a publicly released dataset of anonymized customer resource usage statistics from Microsoft's Azure SQL Database production system over a three-month period. We implemented the STeP framework to ingest and analyze this large dataset. STeP allowed us to use this production dataset to evaluate several new algorithms for packing database tenants onto servers. These techniques produce highly efficient packings by collocating tenants with compatible resource usage patterns. The evaluation shows that under a production-sourced customer workload, these techniques are robust to variations in the number of nodes, keeping performance objective violations to a minimum even for high-density tenant packings. In comparison to the algorithm used in production at the time of data collection, our algorithms produce up to 90% fewer performance objective violations and save up to 32% of total operational costs for the cloud provider.

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.

SoCC '16, October 05 - 07, 2016, Santa Clara, CA, USA.
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4525-5/16/10. . . $15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987575

Keywords Database-as-a-Service, Cloud database, Multi-tenancy

1. Introduction

As many companies and organizations move their services to the cloud, cloud providers must manage computing resources at a larger scale than ever before. Several cloud providers, including Microsoft and Amazon, got their start by offering application-independent computing resources, or Infrastructure as a Service (IaaS). The same providers have also moved to the Platform as a Service (PaaS) model, wherein specific applications are rented rather than raw computing resources. This paper addresses an important subset of PaaS offerings: the Database-as-a-Service (DBaaS) model [19]. More specifically, this paper addresses the question of how to assign DBaaS workloads to hardware resources, and we examine this issue in the context of Microsoft's Azure SQL Database (ASD) product.

There has been a great deal of industrial and academic interest in cloud database systems. From a user's perspective, these systems are appealing because they support simple deployment, elastic scaling, and transparent administration [4]. Providers seek to maintain an economically viable service by making use of *multitenancy*, or serving multiple customers from a single server. A significant amount of research examines the problem of multitenant packing (or allocation), but the prior work focuses on relatively small scale problems containing hundreds of tenants and tens of servers. These scenarios are well known in private clouds, but they are not representative of multitenancy at scale.

In this paper we examine "where the rubber meets the road" for DBaaS offerings. Our focus is large-scale public cloud offerings, where millions of tenants are the norm and tenant density per machine is multiple orders of magnitude higher than studied in prior research. Here, the opportunities for cost savings are enormous. For example, James Hamilton

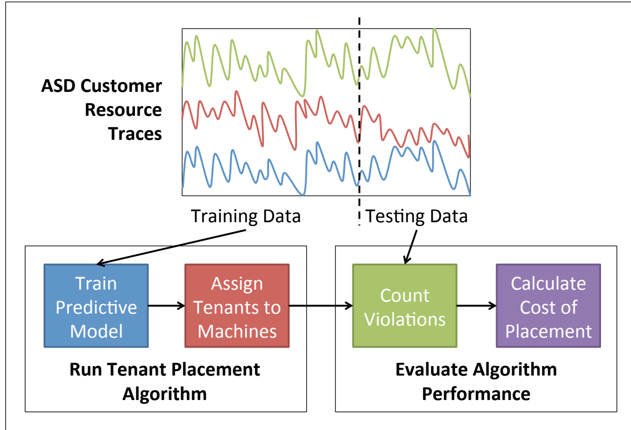


Figure 1: The STeP Framework

said that Amazon serves tenants for 2-5 times lower cost than conventional in-house operations [20]. Microsoft boasts similar economic efficiencies from working at scale; there are currently more than one and a half million customer databases deployed on Microsoft's ASD service [21]. While large-scale cloud providers drive down their hardware and infrastructure costs by bulk acquisition and cheap energy sources [18], the actual software system running the service has a huge impact on how well those physical resources are used. In this work, we focus on the tenant allocation strategy, examining how well different strategies work when they are applied to real-world customer traces from ASD.

As part of this analysis we present **STeP** (Scalable Tenant Placement), a framework for increasing the efficiency of large-scale, public DBaaSs by maintaining a healthy user density in the cluster through intelligent collocation of tenants. As we discuss in Section 2, the multitenancy trade-off between low density of users on a machine (providing a higher likelihood of good performance) versus cost is well understood. However, none of these prior studies have analyzed placement algorithms in the context of real-world, production-scale user behavior.

In the STeP framework, we evaluate several approaches to tenant packing using recently released, production-sourced resource traces from ASD over a three month period in late 2014 [21]. Our study is the first to use these real-world, production-scale resource traces with hundreds of thousands of databases in the evaluation of tenant placement algorithms. This aspect of STeP sets us apart from the vast majority of prior tenant packing studies which relied on well-defined benchmarks to simulate cloud workloads. The real-world traces reveal that, in contrast to synthetic benchmark workloads, individual user workloads are dramatically diverse and exhibit extreme swings in resource consumption.

A placement algorithm is successful if it is able to produce *high quality* tenant packings. In this study, our measure of quality is defined by the number of performance objective violations per tenant based on the ASD resource traces. In addition, we evaluate the business costs associated with a tenant placement decision using two different metrics: "MSFT" and "PMAX". These metrics capture the provider's cost of operation (e.g., hardware prices), as well as the monetary penalty associated with violations of a user's performance objectives. All of our quality measures are described in detail in Section 4.

As part of the STeP framework, we test four scalable tenant placement algorithms: Scalar Static, Summed Time Series, FFT Covariance, and Dynamic. These four algorithms leverage predictive models to project per-tenant resource usage, and they collocate Azure databases that are likely to exhibit compatible access patterns. Features used by the predictive models include summary statistics, time-series shapelets, FFT coefficients, and dynamicity. The algorithms consist of variations on classic greedy bin packing approaches and hierarchical clustering techniques. Many of these ideas have been applied in other domains, but we combine them in novel ways. To the best of our knowledge, we are the first to apply several of these ideas to the tenant placement problem. The algorithms are described in detail in Section 5.

Fig. 1 summarizes the STeP framework. As depicted, STeP uses a subset of the ASD data to train a predictive model and run one of several tenant placement algorithms to assign tenants to servers. Then it uses the remainder of the ASD data to evaluate the performance of the algorithm by counting performance objective violations and calculating the overall cost of the placement.

Using the quality measures mentioned above, our results in Section 6 demonstrate that all of our placement algorithms improve over Microsoft's basic algorithm. First, we show that the configuration produced by our simplest algorithm, the Scalar Static algorithm, costs up to 26% less than the best configuration produced by the basic algorithm. As our algorithms increase in the number of parameters and complexity, we show that further cost reductions are possible. Specifically, the Summed Time Series and Dynamic algorithms cost up to 11% less than the Scalar Static algorithm, and the FFT Covariance algorithm costs up to 8% less than the Scalar Static algorithm. All of these algorithms achieve their cost savings by reducing performance objective violations–they improve by up to 90% in comparison to the basic algorithm. Lastly, we also show that a classic approach to tenant packing, the "Best Fit" greedy algorithm, does not perform well on this real-world dataset. Our evaluation shows that this approach performs over an order of magnitude worse than Microsoft's basic algorithm.

The contributions of this paper are:

- The first study on characterizing tenants for multitenant placement by employing a framework that analyzes a real production-scale trace of user behavior in a cloud database service.

- Four scalable tenant placement algorithms that look at different aspects of the workload time-series: summary statistics, time-series shapelets, FFT coefficients, and dynamicity.
- Two comprehensive DBaaS cost models based on the cloud provider's operational costs and penalties for performance objective violations.
- Extensive experimental results demonstrating that our techniques cause up to 90% fewer performance objective violations and save up to 32% of operational cost in comparison to existing approaches.

2. Related Work

There is broad research on database multitenancy, including tenant scheduling and placement [9, 25], live migration of tenants [11, 14], and even the architecture that will best facilitate multitenancy [10, 23]. In this paper we focus on a subset of this space, namely the modeling and placement of tenants on a shared-nothing cluster of database servers at scale.

2.1 Multitenancy Models

There are many architectures to enable database multitenancy, but three have proved dominant in practice. The *shared hardware* model enables multitenancy through the use of virtual machines (VMs) to isolate tenants from each other, providing manageability at the expense of poor tenant density and uncoordinated resource utilization [7, 31]. The *shared table* model is another approach where all of the tenants' data is collocated in the same database and tables [1, 30]. This model provides a higher level of consolidation due to minimal redundancy and catalog metadata per tenant, but at the cost of complexity and poor resource isolation. Striking a balance between these models, and the focus of this paper, is the *shared process* model. Here many tenants run in a single database process and share resources such as a log writer or buffer pool space.

With this *shared process* model, there are two common techniques to ensure tenants meet their performance objectives: resource allocation through throttling and scheduling [27], and soft isolation through smart workload colocation [26]. The main risk with soft isolation is that tenants may be starved for resources if given a poor initial placement or if the database workloads change. The benefit is that tenants can opportunistically use resources as needed.

2.2 Tenant Placement Strategies

There has been considerable interest in modeling DBaaS workloads and translating these approximations into tenant placement plans. Each of the prior solutions differs in key ways from our approach.

Kairos seeks to minimize the number of servers required to host a static number of tenants by smart memory management [8]. This strategy is expensive for large datasets, so they target deployments with hundreds of tenants on tens of servers. Our research addresses a setting that is much larger in scale.

Pythia uses a complex set of features to classify database tenants and to determine which combination of tenant classes will result in a performance violation [15]. Similar to Kairos, Pythia is evaluated at a smaller scale, with hundreds of tenants on tens of servers. It uses a randomized set of synthetic benchmarks for its workload.

PMAX addresses tenant placement with the optimization of cost-driven performance objectives [24]. Their technique differs from ours in that it uses an expensive dynamic programming algorithm and is evaluated using a single synthetic workload.

RTP [28] focuses on read-mostly and main-memory workloads, allowing tenants to be load-balanced between several servers and resilient to node failures. While their workloads are based on real data from SAP, the unsampled Azure dataset we use contains three orders of magnitude ($> 100K$) more tenants.

Lang et al. present a study that looks to simultaneously address tenant placement concurrently with server configuration [22]. Their study relies on the TPC-C benchmark as a workload, while our research focuses on finding solutions that can handle the real-world variability in the Microsoft production traces.

Quasar [13] and its predecessor Paragon [12] focus on general cloud workloads, not database-specific workloads. They classify workloads by scale-out, scale-up, and heterogeneity, and identify which server configurations will perform best on the class.

Floratou and Patel present RkC to address replica placement in a multitenant environment using a limited random subset of the cluster and the TPC-C benchmark [17]. While they demonstrate that randomized placement strategies can perform well under these constraints, we show in Section 6 that predictive modeling approaches can outperform randomized placement when evaluated on *real workloads*.

3. Microsoft Azure SQL Database

In this paper, we analyze a publicly available telemetry dataset from Microsoft's Azure SQL Database (ASD) [21]. An in-depth description of the ASD system is available in [2, 3]. We now highlight the features of the ASD dataset that are most salient to this study.

Microsoft's ASD allows users the pay-as-you-go capability to scale up or out depending on their relational data processing needs. Microsoft operates ASD clusters in 20 regions around the world and each is of varying size, network, and server capacity. We studied two cluster telemetry traces from North America and Europe spanning three months in 2014. Each data center provided a trace for a single cluster of 100–150 hosts serving hundreds of thousands of ASD instances. There are many ASD clusters hosted in a data

center, but customer databases rarely move across clusters. Therefore to study long-term behavior, we deemed a single cluster's trace (in each data center) sufficient. In ASD, multiple tenants are housed in one physical SQL Server database in a single process instance. Our techniques use soft isolation–collocating tenants with complementary resource usage patterns–to manage resource allocation.

Finding an efficient assignment of tenants to servers is complex because tenants are heterogeneous in both their service level objectives and their observed resource usage. The ASD subscription model consists of three tiers (*Basic*, *Standard*, and *Premium*) offering varying performance, availability and reliability objectives, among other differences [2, 21]. The data shows, however, that a tenant's observed resource usage does not always correspond to the expected usage for their subscription class. Further complicating this issue, the physical needs of individual tenants evolve over time. In this study, we examined the resource usage of all of the tenant databases to propose a multitenant placement strategy based on the *actual* resource utilization patterns instead of the conservative minimum resource guarantees.

In the two clusters we studied, to satisfy availability objectives, a customer database has a k-way replication factor with $k = 3$ by default. In ASD, one replica is deemed the "primary" replica and the other replicas are marked as "secondary" replicas. In lieu of migrating tenants for load balancing, ASD replicas may be promoted to primary status through a "swap" operation whereby the old primary becomes a replica.

In addition to swaps, the ASD service may also physically migrate a replica from one server to another. This is an expensive, but sometimes unavoidable, operation. When replicas are moved (or initially placed), the placement adheres to a set of upgrade and fault domain placement (server grouping) policies. Essentially, replicas must be distributed over different hosts in different upgrade and fault domains. These policies add another layer of complexity to this applied tenant placement problem.

We analyze the ASD traces to understand and characterize the resource utilization of tenants in a public cloud setting. By identifying differences between the behavior of user databases, such as variations in the number of logical page reads, we hope to identify strategies for efficiently packing tenants on hosts. Our hypothesis is that training predictive models on these production traces will result in tenant assignments that are more efficient and robust to load spikes than those produced by ASD's basic algorithm.

3.1 Telemetry Dataset

The traces contain a set of anonymized time series readings that capture the resource utilization of each customer database, including its replicas. ASD records the utilization rates across a wide range of resources. We focus on four resources: *cpu_time*, *logical_reads_pages*, *logical_writes_pages*, and *log_operations*. The values of these record fields represent the summed observed resource utilization for a time window. The schema of the data is shown in Table 1.

Attribute	Description
timestamp	The time when this window ends.
interval_seconds	The duration of this time window (when the window starts).
dbuid	The anonymized unique database id.
replica_type	The type of the replica: primary, secondary, or transitioning.
machine_id	An id denoting the machine that this replica resides on during the window.
max_worker_count	The maximum number of worker threads available to this database.
cpu_time	The CPU time spent by this database on this machine in a time window (in microseconds).
logical_reads_pages	The number of logical pages accessed in a given time window.
logical_writes_pages	The number of logical pages written in this time window.
log_operations	The number of log operations performed in this time window. i.e., transaction appends to the write-ahead log

Table 1: Schema of the ASD Performance Telemetry Dataset [21]

Each record in the time series represents the resource utilization of a database replica for a time interval defined by the end time of the window (*timestamp*) and the length of the window (*interval_seconds*). The target length of the telemetry records is 300 seconds, but in practice the length of these recordings varies significantly. The subscription class of the database (e.g., Basic, Standard, Premium, etc.) can be inferred by the *max_worker_count* field as this lists the maximum number of worker threads available for query processing. A large *max_worker_count* indicates that the database belongs to a premium-tier customer. In this 2014 dataset, 92% of customers have a *max_worker_count* of 180. The remaining 8% range from 0 to 1600.

3.2 STeP Data Pre-Processing

As described in the previous section, each record in the dataset represents a single replica's activity during a time interval defined by the *timestamp* and *interval_seconds* fields. Ideally, this time interval would correspond to a five-minute window perfectly aligned to one of the twelve non-overlapping five-minute slots per hour. If this were the case, we could easily determine how much load was on a given machine at any given time (a necessary calculation for evaluating the performance of a tenant placement algorithm).

In this study we transform the raw dataset to align its records with evenly spaced five-minute windows. To do so, we split each record at the boundaries between time windows, and expand the time interval of each resulting record to fill a full window. For simplicity, we assume that resource utilization is uniform throughout each record's dura-

timestamp	interval_seconds	cpu_time
10:08:00	300	10000
10:14:50	350	20000

↓

timestamp	interval_seconds	cpu_time
10:05:00	300	4000
10:10:00	300	6000
10:10:00	300	3428.57
10:15:00	300	16571.43

↓

timestamp	interval_seconds	cpu_time
10:05:00	300	4000
10:10:00	300	9428.57
10:15:00	300	16571.43

Figure 2: Example STeP pre-processing pipeline for a single database replica on one machine. For simplicity we show only three of the ten fields in each record.

tion. Therefore, if a measurement spans multiple windows, each window gets the fraction of the measurement for which it has coverage.

This transformation has the potential to double or triple the size of the dataset by splitting each record into two or more records. So for the final step, we aggregate all rows for each replica for each time window into a single row. See Fig. 2 for a simple example of the data pre-processing pipeline.

4. Tenant Placement

We set out to analyze several different algorithms to pack databases onto machines. Tenant placement must be efficient, but also ensure that users' resource requirements are met. We define a "violation" of these requirements as a moment in time when a machine is consuming a particular resource in aggregate (either logical reads, logical writes, CPU usage, or write-ahead log operations) above some threshold consumption level. Each time a machine has a violation, we assume that each of the active tenants on that machine may have a service-level agreement (SLA) violation.

Microsoft pays a penalty for service outages to its customers (both immediate monetary penalties and long-term reputation penalties), so the company has a clear interest in minimizing these incidents. Additionally, each server costs some amount of money per minute to operate (including hardware, power, and administration costs), so all cloud providers have an interest in minimizing the number of machines deployed to serve their customers. Our algorithms are designed to minimize the sum of a cloud provider's operating expenses and penalties paid for service-level violations.

A secondary goal of our study is to find a stable packing of database tenants onto machines so that migration of tenants is rarely necessary. Migrations can increase server utilization and have the potential to disrupt the tenant's workload, so it is best to avoid this overhead whenever possible.

Symbol	Definition
MAX_READ	Maximum summed logical reads ever seen on a single machine in the dataset
MAX_WRITE	Maximum summed logical writes ever seen on a single machine in the dataset
MAX_CPU	Maximum summed CPU utilization ever seen on a single machine in the dataset
MAX_LOG	Maximum summed log operations ever seen on a single machine in the dataset
P	Percentage of the observed maximum for any resource above which a machine is considered to be in violation (either 0.65 or 0.85)
$in_violation(m,t)$	Returns a binary variable indicating whether or not server m is in violation at time t
$READ_{d,t}$	Logical reads of database d at time t
$WRITE_{d,t}$	Logical writes of database d at time t
$CPU_{d,t}$	CPU usage of database d at time t
$LOG_{d,t}$	Log operations of database d at time t
$is_active(d,t)$	Returns a binary variable indicating whether or not database d is active (has a non-zero value for some resource) at time t
D	The set of all databases in the dataset
$D_{m,t}$	The set of active databases assigned to machine m at time t
S_P	Constant scale factor to give "PMAX" violation penalty a proper weight relative to the server cost
S_M	Constant scale factor to give "MSFT" violation penalty a proper weight relative to the server cost
y	Number of servers available for tenant placement
G	The cost per server per month
w_d	The *max_worker_count* of database d
N	Number of time slices in the testing period
Q	Number of months in the testing period
N_q	Number of time slices in month q
$m_{d,t}$	The machine hosting database d at time t
$v_{d,q}$	The % downtime for database d in month q
$p(v_{d,q})$	The penalty for downtime $v_{d,q}$ for "MSFT"

Table 2: Symbols used in tenant placement evaluation.

For convenience, we provide a glossary of all the symbols used throughout Section 4 in Table 2.

4.1 Service-Level Agreement Violations

In order to evaluate the efficacy of our tenant placement algorithms, we must first define the conditions under which a service-level violation may arise. Our dataset does not record whether a particular tenant actually experienced a SLA violation under some configuration of the system, so instead, *we take the pessimistic view that any tenant with activity on a machine that has high total utilization of any resource is experiencing a violation.*

So how do we define "high utilization"? Internally, a DBaaS provider typically aims to keep total resource usage (e.g., summed CPU time) of a machine's customer databases under 65% (and 85%) of the maximum possible value.[1] We

[1] We use 65% and 85% thresholds to allow us to account for the additional load of secondary replicas, background service load, and essential system load. The latter two load parameters are proprietary.

do not know Azure's real hardware specifications (they are proprietary). For CPU time, we instead bound this value by the maximum summed CPU utilization ever seen on a single machine in the dataset (we will call this *MAX_CPU*). We do the same for the other three resources (defining the values *MAX_READ*, *MAX_WRITE*, and *MAX_LOG*) since we also do not know the true maximum for these resources.

As *MAX_READ*, *MAX_WRITE*, *MAX_CPU*, and *MAX_LOG* are lower bounds on the true maximum, we decided to examine two different thresholds: 65% and 85% of the observed maximum for each resource. Given a percentage P of the observed maximum (either 0.65 or 0.85 depending on the chosen threshold), a **violation**, or set of violations, will occur on machine m at time t with databases $D_{m,t}$ if the following is true:

$$\begin{aligned} in_violation(m,t) = & \left(\sum_{d \in D_{m,t}} \text{READ}_{d,t}\right) > P * \text{MAX_READ} \vee \\ & \left(\sum_{d \in D_{m,t}} \text{WRITE}_{d,t}\right) > P * \text{MAX_WRITE} \vee \\ & \left(\sum_{d \in D_{m,t}} \text{CPU}_{d,t}\right) > P * \text{MAX_CPU} \vee \\ & \left(\sum_{d \in D_{m,t}} \text{LOG}_{d,t}\right) > P * \text{MAX_LOG} \end{aligned} \tag{1}$$

where $READ_{d,t}$, $WRITE_{d,t}$, $CPU_{d,t}$ and $LOG_{d,t}$ represent the logical reads, logical writes, CPU usage, and log operations of tenant d at time t. A violation is recorded if any one of these resources exceeds its threshold.

4.2 Tenant Placement Efficacy

Given this definition of a violation, we now define three metrics to quantify the effectiveness of a tenant placement strategy:

1. **Violations:** Minimize the total number of tenants experiencing violations per hour, given y available machines.
2. **"PMAX":** Minimize the total cost of system operation per month, using a cost model inspired by the work of Liu, et al. [24]. Details below.
3. **"MSFT":** Minimize the total cost of system operation per month, using a cost model based on Azure's real penalty model [2]. Details below.

"PMAX": The second metric attempts to capture the trade-off between high customer satisfaction and low cost of ownership for the cloud provider. Similar to Liu, et al., we define the total cost of an allocation with y machines as the operating expense of running y servers plus the sum of any penalties arising from SLA violations. In keeping with the PMAX approach, we weight our penalties by the customer's expected level of service. We approximate this figure using the tenant's *max_worker_count*, because it is a proxy for the customer's subscription class as described in Section 3.1. Our model presumes that violations for high-paying customers are more costly. The SLA penalty in Liu, et al. is between 2 and 20 units by default [24], so to create a similar penalty we divide the *max_worker_count* by 10 (as described in Section 3.1, most customers have a *max_worker_count* of 180). We also multiply the total SLA penalty by a constant scale factor S_P in order to give it a proper weight relative to the server cost. As detailed in Section 6.1, we define S_P based on the scale factor used by [24]. Therefore, the cost per month is:

$$cost = y \times G + \frac{S_P}{Q} \sum_{t=1}^{N} \sum_{m=1}^{y} (in_violation(m,t) * \sum_{d \in D_{m,t}} \frac{w_d}{10}) \tag{"PMAX"}$$

where y is the number of servers, G is the cost per server per month, Q is the number of months in the testing period, N is the number of time slices in the testing period, $in_violation(m,t)$ returns a binary variable indicating whether or not server m is in violation at time t in accordance with Eq. (1), $D_{m,t}$ is the set of active databases (databases with a non-zero value for some resource) assigned to machine m at time t, and w_d is the *max_worker_count* of database d.

"MSFT": The third metric is another attempt to navigate the trade-off between reducing SLA violations and minimizing operating expenses, but the SLA violation cost varies depending on the total number of violations each tenant experiences in a month. Specifically, Microsoft incurs a penalty each month for each customer experiencing an availability outage more than 0.1% of the time. Furthermore, the penalty is 2.5 times higher if the customer experiences an outage more than 1% of the time. Thus, the cost per month for the "MSFT" metric is:

$$cost = y \times G + \frac{S_M}{Q} \sum_{q=1}^{Q} \sum_{d \in D} p(v_{d,q}) \tag{"MSFT"}$$

given the percentage downtime for database d in month q is:

$$v_{d,q} = \frac{1}{N_q} \sum_{t=1}^{N_q} (in_violation(m_{d,t},t) * is_active(d,t))$$

and the penalty for downtime is:

$$p(v_{d,q}) = \begin{cases} 0 & \text{if } 0 \le v_{d,q} < 0.1\% \\ 1 & \text{if } 0.1\% \le v_{d,q} < 1\% \\ 2.5 & \text{if } v_{d,q} \ge 1\% \end{cases}$$

where D is the set of all databases, N_q is the number of time slices in month q, S_M is the scale factor (to be defined in Section 6.1), $is_active(d,t)$ returns a binary variable indicating whether or not database d is active (has a non-zero value for

some resource) at time t, and $m_{d,t}$ is the machine hosting database d at time t. All of the other variables reflect their definitions from the "PMAX" metric.

The main difference between the last two metrics is that the "PMAX" penalty varies linearly with violations per tenant, while the "MSFT" penalty varies according to a step function; it places an upper bound on the penalty per tenant and allows small numbers of violations per tenant with no penalty. Additionally, "PMAX" charges a higher penalty for SLA violations to high-paying customers. Although "MSFT" more accurately models the real penalty used by Microsoft, we include "PMAX" as well since this is the model used by [24], and other cloud providers may prefer this linear cost model.

5. Tenant Placement Algorithms

Our first set of algorithms generate a **static** allocation of databases to machines. By static, we mean that once a database is assigned to a machine, it will not change for the remainder of the three-month period. We also examine **dynamic** algorithms that adjust database placement in an effort to react to unforeseen changes in behavior.

5.1 Static Allocation

Our baseline allocation algorithm randomly assigns databases to machines. In this algorithm, each database is assigned to one machine, chosen uniformly at random from the set of available machines. This strategy is similar to the one used by Azure in production clusters in late 2014. We use our **Random** placement algorithm as the baseline approach.

As we will show in Section 6.2, the random placement approach actually performs quite well compared to other more complex allocation strategies, due in part to the extremely high database density on the machines. The question is: can we do better?

An alternative approach is to use some portion of the data (e.g., the first n weeks of our dataset) to train a model. Then we can use this model to assign databases to machines and test the performance of our algorithm on the rest of the data. All of the algorithms that follow use this approach.

For convenience, we provide a glossary of the symbols used throughout Section 5 in Table 3. Table 2 defines the symbols that also appear in Section 4.

5.1.1 Predictive Greedy Algorithms

The following set of algorithms use a greedy approach in which we assign databases to machines one at a time based on one of two cost models: a *Scalar* model or a *Summed Time Series* model. In each case, we train the model by assigning a cost to each database based on its CPU usage data during the training period. We also define how placing a database on a machine affects the load on that machine.

Scalar Cost Model: For our first greedy algorithm, we use a simple cost model based on the average CPU usage

Symbol	Definition
$cost_{d,z}$	The cost of each database d in the Scalar model during the training period z
$base_{d,z}$	The base cost of each database d during the training period z (for the Scalar model)
$a(w_d, z)$	The additional cost of each database d during the training period z based on *max_worker_count* w_d (for the Scalar model)
$N_{d,z}$	Number of time slices in the training period z in which database d exists
D_z	The set of all databases with activity during the training period z
D_m	The set of all databases currently assigned to machine m
$L_{m,z}$	Load on machine m during the training period z
$\{z_1..z_k\}$	The k segments of the training period z (for the Summed Time Series model)
x	The number of "large" databases (the top 10% of databases based on Eq. (2))

Table 3: Symbols used in placement algorithms. Contains symbols carried over from Table 2.

for each database during the training period. Given the CPU training data, we define the "cost" of each database d during the training period z as:

$$cost_{d,z} = base_{d,z} + \lambda * a(w_d, z) \quad (2)$$

with a base cost, $base_{d,z}$ of:

$$base_{d,z} = \frac{1}{N_{d,z}} \sum_{t=1}^{N_{d,z}} (CPU_{d,t})$$

and an additional cost, $a(w_d, z)$ of:

$$a(w_d, z) = \begin{cases} \underset{s \in D_z, w_s = w_d}{\text{avg}} (base_{s,z}) & \text{if } \exists s \in D_z \text{ s.t. } w_s = w_d \\ \underset{s \in D_z}{\text{avg}} (base_{s,z}) & \text{otherwise} \end{cases}$$

where $N_{d,z}$ is the number of time slices in the training period in which database d exists[2], $CPU_{d,t}$ represents the CPU usage of database d at time t, D_z is the set of all databases with activity during the training period, and w_d is the *max_worker_count* of database d (see Section 3.1). We add some additional cost $a(w_d, z)$ to each database's base cost in order to account for the fact that some databases have no telemetry data or very little telemetry data during the training period due to a lack of activity. The additional cost is a function of the *max_worker_count* since customers who have paid for the same level of service are more likely to have similar resource needs than two unrelated customers. We empirically found that $\lambda = 0.5$ appropriately weighted the additional cost relative to the base cost.

For simplicity, we only use the CPU data to train our models. The cost model for testing the allocation continues to use all four resources, as described in Eq. (1).

[2] The dataset does not include information about when each database was created or dropped, so we make the conservative assumption that each database only exists between the first and last time slices in which it has activity during the three month period.

Using a simple greedy bin packer algorithm, we attempted to evenly distribute the summed cost of the databases across the machines. This algorithm works by repeatedly placing the largest (highest cost) database replica not yet allocated onto the current emptiest (least loaded) machine that satisfies the upgrade and fault domain constraints (i.e., the machine is not part of any upgrade or fault domains already containing previously allocated replicas of the given database; see Section 3). We assume that CPU cost is additive [8, 28], so the load on a machine is defined to be the sum of the cost of all databases assigned to it so far.

There is an element of randomness in this greedy algorithm because if two databases have the same cost, either one may be allocated first, resulting in two different allocations of databases to machines. The number of possible allocations explodes if there are many databases without training data, since all of these databases that have the same *max_worker_count* will have the same cost, and can therefore be allocated in any order.

This greedy algorithm is different from the "Best Fit" greedy algorithm and Integer Linear Programing (ILP) bin packing algorithms used in other work [8, 24]. While a "Best Fit" or ILP approach tends to group the largest databases together on a few machines, our algorithm (unfortunately known as "Worst Fit" in other applications [6]) ensures that the large databases are as dispersed as possible across all available machines. To show the benefit of our approach, we implemented a "Best Fit" algorithm that sorts databases by decreasing cost and iteratively places each database on the *most loaded* machine that it fits on (satisfying upgrade and fault domain constraints). In Section 6.2, we show that this "Best Fit" approach creates an allocation that performs significantly worse than random.

Summed Time Series Cost Model: As described above (and as we will show in Section 6.2), it is possible to create a very good allocation of databases to machines using a simple cost model with a scalar value for each database. But can we do even better by taking advantage of the training data at a finer granularity?

To answer this question, we extended our greedy model to consider the time series of resource usage rather than scalar aggregates. This approach is optimized to collocate databases only if their resource usage time series are anti-correlated. For example, if one tenant is especially active during the day, it would be wise to place that database on a machine with another tenant that is active at night to avoid concurrent spikes in resource usage.

The Summed Time Series model builds upon the Scalar model described above, but it makes allocations using a time series of resource usage. As in the Scalar model, the Summed Time Series model iterates over the databases in order of decreasing size (as defined by Eq. (2)), assigning them to machines one at a time. Instead of placing each database on the currently least loaded machine, however, this algorithm examines how the database's time series of CPU usage would affect the summed resource utilization at each time slice on each available machine. Then it chooses the machine on which the maximum value of the summed time series is the smallest (satisfying upgrade and fault domain constraints).

To avoid overfitting and to make this problem tractable, we aggregated each time series into larger segments (e.g., 6 hours or more) and found the cost from Eq. (2) over each segment. This gives us a new definition of the load per machine for the Summed Time Series cost model. If the training period z is split into k segments $\{z_1..z_k\}$ (each segment may have multiple time slices), the load on machine m during training period z is:

$$L_{m,z} = \max_{i=1}^{k}\left(\sum_{d \in D_m} (cost_{d,z_i})\right) \quad (3)$$

where $cost_{d,z_i}$ is as defined in Eq. (2), and D_m is the set of all databases currently assigned to machine m.

5.1.2 FFT Covariance Cost Model and Algorithm

In this approach, we compute the fast Fourier transform (FFT) of the CPU usage time series for each of the highly active "large" databases (the top 10% of databases based on Eq. (2)). To avoid overfitting and to make this solution scalable, we then eliminate all but a fraction of the FFT coefficients for each large database; this ability to model a time series with only a few coefficients is a key advantage of using the FFT.

In order to determine which databases to place together on a machine, we perform agglomerative hierarchical clustering with complete linkage [16]. Hierarchical clustering works by iteratively combining items with minimum "distance" into a hierarchy. Here, we define the "distance" between databases as the pairwise covariance of the truncated FFT. This results in a hierarchy in which anti-correlated databases will likely be near each other, since anti-correlated databases have a low pairwise covariance value and thus a smaller "distance" between them.

Once we have our hierarchy, we use it to perform hierarchically structured bin packing [5] in order to create clusters of the desired size. Ideally we will have one cluster per machine, so we try to have clusters of size x/y, where x is the number of large databases and y is the number of machines.

This algorithm is summarized in Alg. 1. Note that we only perform this analysis on the highly active primary database replicas, as pairwise covariance and clustering of all databases would be extremely computationally intensive. We allocate the remaining "small" databases using the Scalar greedy algorithm from Section 5.1.1.

5.2 Dynamic Allocation

In the Azure cloud environment, workloads change over time and databases are constantly being created and dropped due to customer churn. These changes and additions could

Algorithm 1: The FFT Covariance Cost Model Algorithm

Input: time series of CPU usage for each of the x large databases; cost for each of the large databases from Eq. (2); F size of truncated FFT; list of y available machines
Output: unique assignment of large databases to machines
Let *max_cost* be $\left(\frac{1}{y}\sum_{i=1}^{x} cost_{d_i}\right) + \varepsilon$;

```
/* Calculate the truncated FFT of CPU usage for
   all databases                                   */
```

for i=1 **to** x **do**
  Calculate FFT of the time series for database d_i;
  Create array f_i of size $2F$;
  Set f_i to the first F real coefficients of the FFT followed by the first F imaginary coefficients;

```
/* Find the covariance of the truncated FFT for
   all pairs of databases                          */
```

for i=1 **to** x-1 **do**
  for j=i+1 **to** x **do**
    Set $cov_{i,j}$ to the covariance of f_i and f_j;

```
/* Perform agglomerative hierarchical clustering
   with complete linkage; use hierarchy to
   perform hierarchically structured bin packing
   */
```

Set cluster group c_i to $\{\{d_i\}\}$, where $i \in \{1..x\}$;
while *number of cluster groups > 1* **do**
  Find cluster groups c_i and c_j with minimum distance, given that $distance(c_i, c_j) = \max_{d_a \in c_i, r_b \in c_j}(cov_{a,b})$;
  Merge cluster groups c_i and c_j;
  foreach *cluster a in cluster group* c_i **do**
    foreach *cluster b in cluster group* c_j **do**
      if $cost_a + cost_b < max_cost$ **then**
        combine clusters a and b;
  Consolidate clusters in the merged group so each cluster c satisfies $cost_c < max_cost$;

Assign each cluster c in the final cluster group to a different machine;

cause some machines to become overloaded and increase the risk of SLA violations. Thus, it is desirable to have a dynamic solution in which the allocation of databases to machines is resilient to changing workloads. Furthermore, when a new database joins the cluster, it must be initially placed without historical workload data and therefore may be assigned sub-optimally. After sufficient data is collected on the new database, a placement algorithm should be able to move it to a more permanent location.

In a live production system, it is not practical to shuffle all database tenants into new locations each time there is a risk of SLA violations. For this reason, we created a dynamic version of our algorithms in which we limit the number of databases that are allowed to move each cycle.

Starting with a static allocation (see Section 5.1), we roll forward the clock on our dataset and monitor the load on each machine. If any machine becomes overloaded, we could balance the load by: (1) swapping primary replicas on that machine with their corresponding secondary replicas on other under-loaded machines [25], or (2) migrating replicas off that machine to other machines with available resources [14]. The ASD traces do not include data on secondary replicas, so unfortunately, we could only study option (2) dynamic algorithms.

Our dynamic algorithm implements migration (option 2 above), and it limits the number of databases that may move between machines. Starting with a complete allocation of databases to machines and an updated set of training data, we move databases one at a time from the most overloaded machine to the most underloaded machine (satisfying upgrade and fault domain constraints) until either the load is balanced or we have moved the maximum number of databases. We move the most active databases first (as defined by Eq. (2)) in order to balance the load with as few moves as possible.

6. Evaluation

In this section we evaluate our algorithms on real-world ASD traces. We start by detailing our experimental setup. Next we examine the efficacy of our static allocations in comparison to random assignments. After that, we show additional cost reductions with the use of dynamic, adaptive tenant placement.

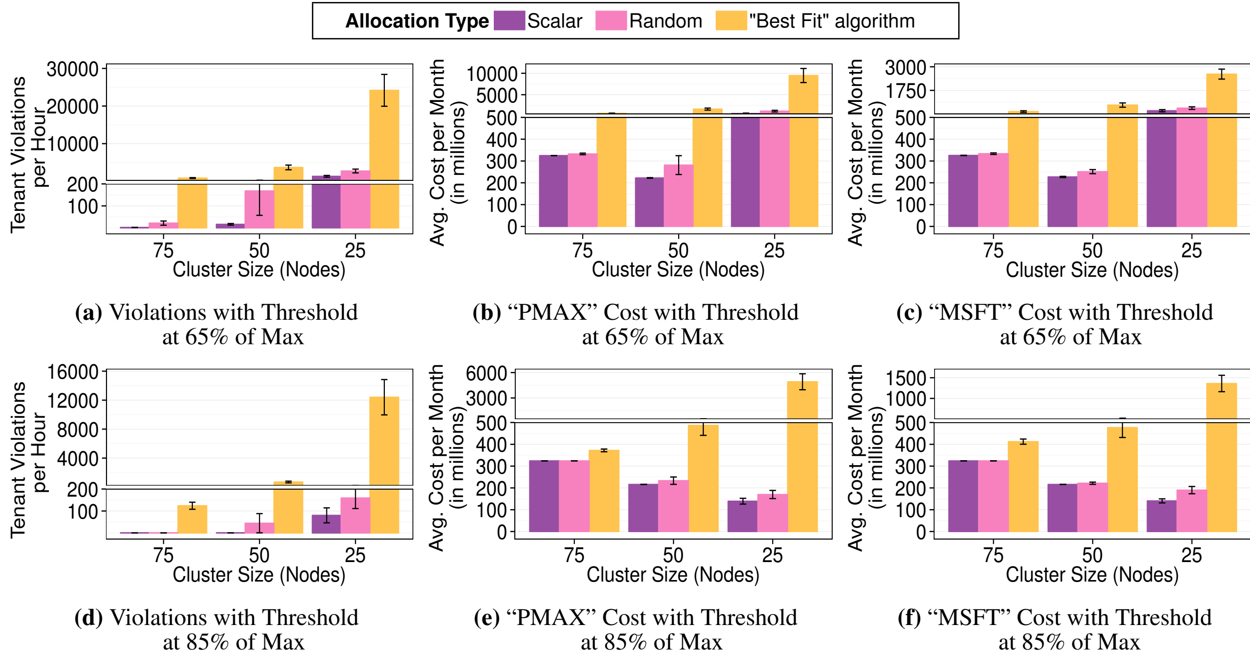
Because there is an element of randomness in all of the algorithms (e.g., not all databases are active during the training period; see Section 5.1.1), we ran each algorithm many times in order to evaluate its effectiveness (the details will be provided as we discuss the results). The metrics used for evaluation are described in Section 4.2.

6.1 Experimental Setup

To calculate the cost of each allocation, we assume that each machine costs 4320000 units per 30-day month, based on the default of 100 units per minute in Liu, et al. [24]. Additionally, to make the ratio between server cost and violation penalty comparable to Liu, et al., we need to calculate the scale factors S_P and S_M in Eqs. ("PMAX") and ("MSFT").

In our "PMAX" model, we count violations at a granularity of 5 minutes, while Liu, et al. count violations on a per-query basis. The tenants in their study submit queries on average every 10 seconds, so each of our violations is equivalent to 30 of theirs (*6 queries per minute * 5 minutes per time slice*). Thus, the scale factor for SLA penalties in Eq. ("PMAX"), S_P, is set to 30.

In Eq. ("MSFT"), the penalty is not a linear function of the number of violations. But for the purpose of comparison, we choose a scale factor based on the penalty for 1%



(a) Violations with Threshold at 65% of Max

(b) "PMAX" Cost with Threshold at 65% of Max

(c) "MSFT" Cost with Threshold at 65% of Max

(d) Violations with Threshold at 85% of Max

(e) "PMAX" Cost with Threshold at 85% of Max

(f) "MSFT" Cost with Threshold at 85% of Max

Figure 3: Tenant violations and operational cost of the Random allocation scheme compared to the Scalar Static allocation scheme and the "Best Fit" algorithm as a function of cluster size in the North American cluster

downtime. This is equivalent to approximately 86 timeslices per month, or 2592 missed query deadlines in Liu, et al. (*0.01 * 8640 time slices per 30-day month * 30 queries per time slice*). We also multiply the SLA penalty for "MSFT" by 18 in order to match the "PMAX" penalty described in Section 4.2. Thus, the scale factor for SLA penalties in Eq. ("MSFT"), S_M, is set to 46656 (*2592 * 18*).

Due to space limitations, we omit the results from the European clusters, but they can be found in an earlier version of this paper [29].

6.2 Static Algorithms

In this section we evaluate the efficacy of several algorithms that produce a static assignment of databases to machines.

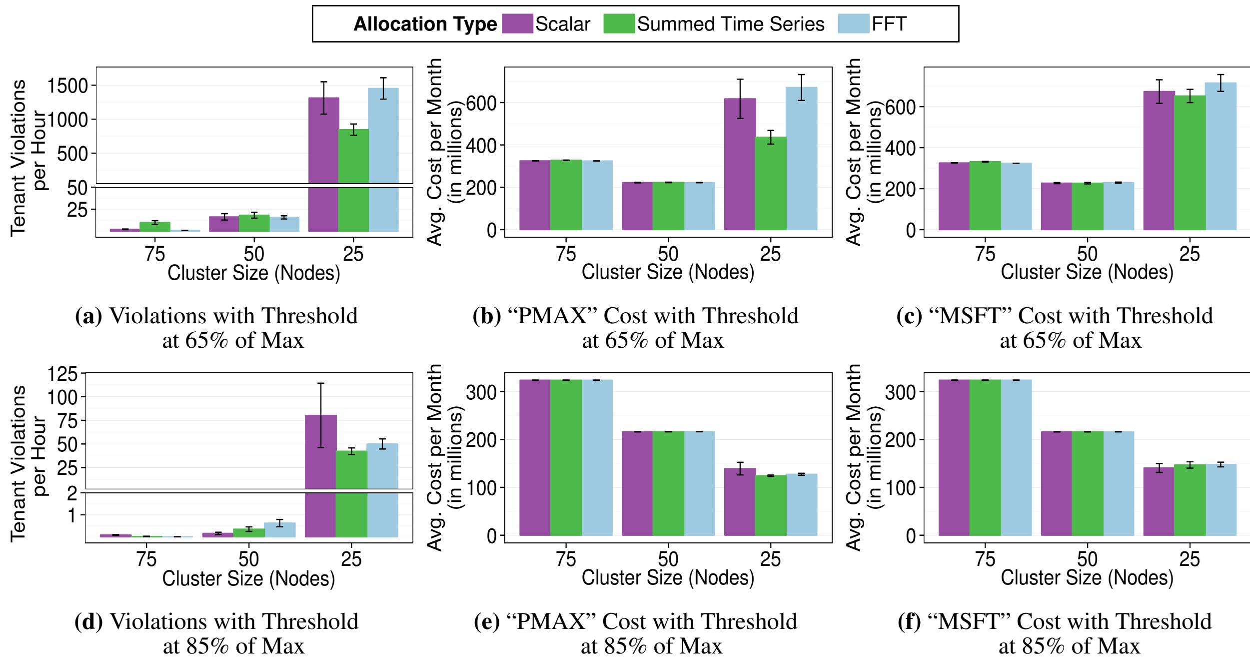
6.2.1 Scalar Cost Model

To see if using a simple predictive model creates an efficient assignment of databases to machines, we compare the Random algorithm to the Scalar cost model greedy algorithms in Fig. 3. These charts show tenant violations and operational cost of the algorithms as a function of cluster size, using the metrics described in Section 4.2. The cluster sizes of 75, 50, and 25 correspond to one half, one third, and one sixth of the original North American cluster of 151 nodes. In these experiments, we train the model on two months and test on the third month of the dataset. We perform cross-validation by testing each of the three months an equal number of times (i.e., train on September and October, test on November; train on September and November, test on October; train on October and November, test on September). *Note the random algorithm does not actually use the training data, but we still test each run on one month for a fair comparison with the predictive algorithms.* In these charts, any machine with resources exceeding 65% (for Figs. 3a to 3c) or 85% (for Figs. 3d to 3f) of the maximum for any resource is considered to be "in violation" (see Section 4.2). Each data point represents the average of 30 independent runs of the algorithm; 10 runs for each of the three months. Error bars show the standard error of the mean.

Fig. 3 shows that it is possible to beat Random by a significant margin with a scalar cost model. In the best case, when using 50 machines, the Scalar cost model produces 90% fewer violations at the 65% threshold than a purely random allocation.

Figs. 3b, 3c, 3e and 3f also show that the Scalar cost model performs better than Random when we consider total operational cost. For the 85% threshold, the Scalar model costs 18% less than Random with the "PMAX" model and 26% less with the "MSFT" model (cost reduction is calculated based on the lowest value for each algorithm, corresponding to 25 machines in each case).

We also tested the performance of the "Best Fit" algorithm with the Scalar cost model from Eq. (2). As shown in Figs. 3a and 3d, the "Best Fit" approach produces over an order of magnitude more violations than a random allocation, especially at reduced cluster sizes. This large number of violations for "Best Fit" translates to a higher penalty cost than Random, and therefore a higher total cost, as shown in the other four charts in Fig. 3. Clearly, the "Best Fit" approach of grouping large databases together does not work given real-world workloads.



(a) Violations with Threshold at 65% of Max

(b) "PMAX" Cost with Threshold at 65% of Max

(c) "MSFT" Cost with Threshold at 65% of Max

(d) Violations with Threshold at 85% of Max

(e) "PMAX" Cost with Threshold at 85% of Max

(f) "MSFT" Cost with Threshold at 85% of Max

Figure 4: Operational cost of the FFT and Summed Time Series allocation schemes compared to the Scalar allocation scheme as a function of cluster size

6.2.2 Summed Time Series Cost Model and FFT Covariance Model

To see if there is any benefit to using a time series instead of a scalar value for the cost of each database, we ran our Summed Time Series algorithm (see Section 5.1.1) with a segment length of one day. We also tested variations on the Summed Time Series model using segment lengths of one week and six hours, but did not find significant differences in performance.

Additionally, we tested our FFT Covariance model (see Section 5.1.2) with the first 62 FFT coefficients corresponding to a minimum wavelength of one day. We also tested variations on the FFT Covariance model in which we truncated the FFT after 10 or 245 coefficients, corresponding to minimum wavelengths of one week and six hours. We did not notice significant differences between these variations.

As with our evaluation of the Scalar cost model, we trained our models on two months of data and tested on the third month, cross-validating across all three months. Fig. 4 shows the performance of these algorithms. As in Fig. 3, each data point represents the average of 30 independent runs of the algorithm; 10 runs for each of the three months.

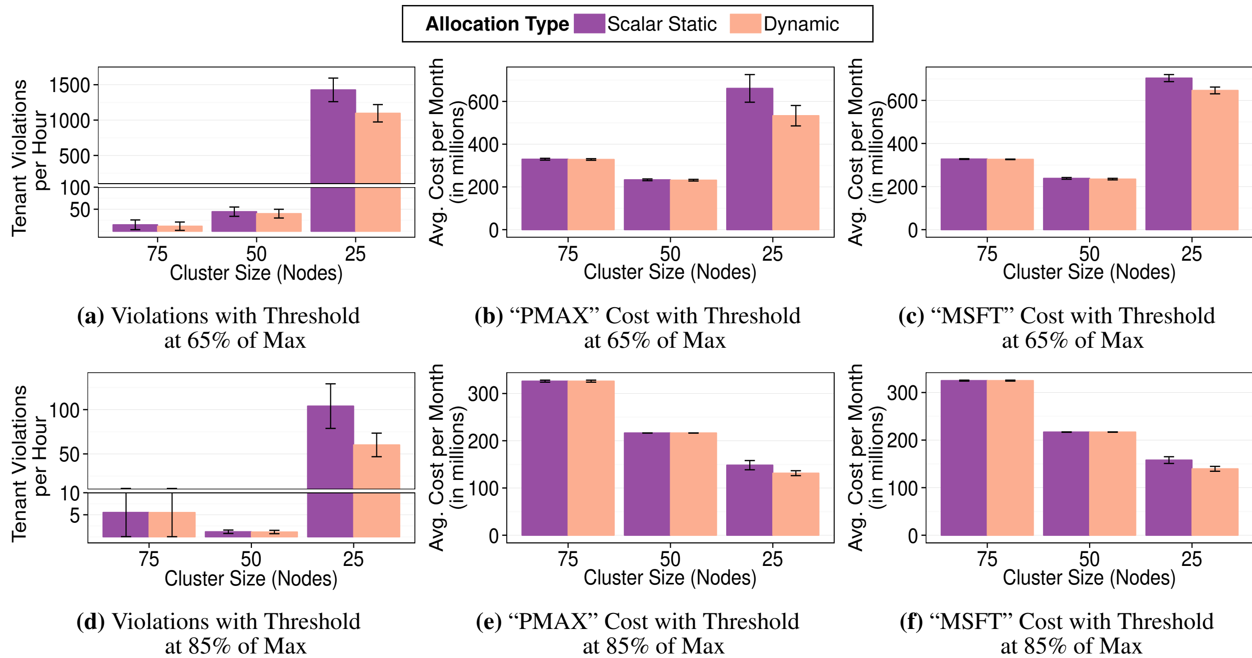
Interestingly, the Summed Time Series and FFT Covariance models do not show significant improvement over the simple Scalar model on this dataset when using 75 and 50 machines. Both algorithms perform as well as or better than Scalar with 25 machines, however, indicating that these methods may have applications for creating denser tenant packings. In particular, the Summed Time Series algorithm caused 47% fewer violations than Scalar and 73% fewer violations than Random (not pictured) at the 85% threshold, leading to 11% lower cost than Scalar and 27% lower cost than Random with the "PMAX" model. The FFT Covariance algorithm showed similar performance at this setting, with 38% fewer violations than Scalar and 69% fewer violations than Random, leading to 8% lower cost than Scalar and 25% lower cost than Random. This sort of dense packing may be especially desirable for a system with expensive hardware or lower penalties for violations. More research is needed to further identify and refine the ideal conditions for each algorithm.

6.3 Dynamic Algorithm

As shown in the previous section, a simple scalar model for static allocation of databases to machines performs very well given our cost model. But if we allow movement of databases, can we do better than a static allocation?

Starting with a static allocation generated with the Scalar model, we evaluated our ability to keep the allocation "up to date" based on changes in the patterns of each database tenant's resource usage. In this experiment, we ran the dynamic algorithm for every week in the dataset, moving a limited number of databases between machines each time.

For the baseline static algorithm, we trained the Scalar model (see Eq. (2)) on the first week of data (September 1 - 7), and tested the resulting allocation on the remaining 12 weeks. Using this allocation as a starting point, we also tested the dynamic model. In this model, we ran the dynamic algorithm once for each week in the dataset. Each time we started with the previous week's allocation and moved



(a) Violations with Threshold at 65% of Max

(b) "PMAX" Cost with Threshold at 65% of Max

(c) "MSFT" Cost with Threshold at 65% of Max

(d) Violations with Threshold at 85% of Max

(e) "PMAX" Cost with Threshold at 85% of Max

(f) "MSFT" Cost with Threshold at 85% of Max

Figure 5: Tenant violations and operational cost of the Dynamic allocation scheme compared to the Scalar Static allocation as a function of cluster size

at most one database onto or off of each machine. Then we tested the performance of the new allocation on the following week. We also tested variations on the dynamic algorithm in which we allowed five or ten databases to move onto or off of each machine during each cycle instead of only one. We did not find a noticeable change in performance by allowing more databases to move per cycle.

Fig. 5 shows the performance of the dynamic algorithm compared to the static algorithm. Each bar is the result of 30 runs of the algorithm. The different metrics are as described in previous sections. In all cases, the dynamic algorithm performs at least as well as the static algorithm, and in most cases outperforms the static algorithm. When using 25 machines, the dynamic algorithm causes 42% fewer violations at the 85% threshold than the static algorithm. Additionally, the dynamic algorithm costs 11% less at the 85% threshold than the static algorithm using the "PMAX" model. Furthermore, both algorithms show significant gains over Random in this experiment (not pictured). At the 85% threshold, Scalar Static costs 23% less than Random and Dynamic costs 32% less than Random using the "PMAX" model. In summary, assuming one migration per machine per week has a negligible impact on cost, our dynamic algorithm provides a significant benefit over the static approaches. We leave for future work an analysis of the costs and benefits of more frequent migrations.

7. Conclusions

In this paper we studied a large, publicly available production dataset from Microsoft's Azure SQL Database. We used this dataset to compare the effectiveness of several tenant placement algorithms, including the algorithm used by Microsoft in production. We analyzed two cost models that take into account server costs and SLA penalties, and used these models to evaluate the tenant placement algorithms. Furthermore, we introduced a predictive greedy algorithm (the Scalar algorithm) to create a static allocation of databases to machines and showed that it can reduce total operational cost by up to 26% compared to a random allocation. We described a dynamic version of this algorithm that enables periodic tenant migration, and showed that it reduces cost by up to 11% compared to the static algorithm.

We also introduced a couple of more complex algorithms that take the time series of CPU usage into account to collocate databases with anti-correlated usage patterns. We showed that these algorithms do not consistently improve upon the Scalar greedy algorithm for low- to medium-density tenant packings, but they show some improvement at high density, causing up to 47% fewer violations for 11% lower cost. In future work we hope to refine these models and identify the ideal conditions for each algorithm.

8. Acknowledgments

The authors would like to thank the reviewers for helping to improve the paper, Nigel Ellis for his support, Frans Kaashoek for providing valuable feedback, James Hamilton for his real-world advice, and Manasi Vartak and Tucker Taft for helpful discussions on modeling the behavior of databases.

References

[1] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *SIGMOD*, pages 1195–1206, 2008.

[2] Microsoft Corporation. `http://azure.microsoft.com/en-us/documentation/services/sql-database/`, 2015.

[3] P. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. Lomet, R. Manne, L. Novik, and T. Talius. Adapting Microsoft SQL Server for cloud computing. In *ICDE*, pages 1255–1263, April 2011.

[4] B. Brynko. NuoDB: Reinventing the database. *Information Today*, 29(9):9–9, 2012.

[5] B. Codenotti, G. D. Marco, M. Leoncini, M. Montangero, and M. Santini. Approximation algorithms for a hierarchically structured bin packing problem. *Inf. Process. Lett.*, 89(5):215–221, 2004.

[6] J. Coffman, E.G., M. Garey, and D. Johnson. Approximation Algorithms for Bin-Packing - An Updated Survey. In *Algorithm Design for Computer System Design*, pages 49–106. Springer Vienna, 1984.

[7] C. Curino, E. Jones, R. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational Cloud: A Database Service for the Cloud. In *CIDR*, 2011.

[8] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan. Workload-aware Database Monitoring and Consolidation. In *SIGMOD*, pages 313–324, New York, NY, USA, 2011. ACM.

[9] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *PVLDB*, 3(1):48–57, 2010.

[10] S. Das, D. Agrawal, and A. E. Abbadi. ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems (TODS)*, 38(1):5, 2013.

[11] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration. *PVLDB*, 4(8):494–505, May 2011.

[12] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *ASPLOS*, pages 77–88, New York, NY, USA, 2013. ACM.

[13] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In *ASPLOS*, pages 127–144, New York, NY, USA, 2014. ACM.

[14] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *SIGMOD*, pages 301–312, 2011.

[15] A. J. Elmore, S. Das, A. Pucher, D. Agrawal, A. El Abbadi, and X. Yan. Characterizing Tenant Behavior for Placement and Crisis Mitigation in Multitenant DBMSs. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 517–528, New York, NY, USA, 2013. ACM.

[16] B. S. Everitt, S. Landau, and M. Leese. *Cluster Analysis*. Wiley Publishing, 4th edition, 2009.

[17] A. Floratou and J. M. Patel. Replica Placement in Multi-tenant Database Environments. In *International Congress on Big Data*, pages 246–253, 2015.

[18] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM*, 39(1):68–73, 2008.

[19] H. Hacigümüş, B. Iyer, and S. Mehrotra. Providing database as a service. In *ICDE*, pages 29–38. IEEE, 2002.

[20] J. Hamilton. private communication.

[21] W. Lang, F. Bertsch, D. J. DeWitt, and N. Ellis. Microsoft Azure SQL Database Telemetry. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 189–194, 2015.

[22] W. Lang, S. Shankar, J. Patel, and A. Kalhan. Towards Multi-Tenant Performance SLOs. In *ICDE*, pages 702–713, 2012.

[23] R. Liu, A. Aboulnaga, and K. Salem. DAX: A Widely Distributed Multi-tenant Storage Service for DBMS Hosting. *PVLDB*, 6(4):253–264, 2013.

[24] Z. Liu, H. Hacigümüş, H. J. Moon, Y. Chi, and W.-P. Hsiung. PMAX: Tenant Placement in Multitenant Databases for Profit Maximization. EDBT, pages 442–453, New York, NY, USA, 2013. ACM.

[25] H. J. Moon, H. Hacigümüş, Y. Chi, and W.-P. Hsiung. SWAT: a lightweight load balancing method for multitenant databases. In *EDBT*, pages 65–76, 2013.

[26] B. Mozafari, C. Curino, and S. Madden. DBSeer: Resource and Performance Prediction for Building a Next Generation Database Cloud. In *CIDR*, 2013.

[27] V. R. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri. SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service. In *CIDR*, 2013.

[28] J. Schaffner, T. Januschowski, M. Kercher, T. Kraska, H. Plattner, M. J. Franklin, and D. Jacobs. RTP: Robust Tenant Placement for Elastic In-memory Database Clusters. SIGMOD, pages 773–784, New York, NY, USA, 2013. ACM.

[29] R. Taft. Predictive Modeling for Management of Database Resources in the Cloud. Master's thesis, MIT, Cambridge, 2015.

[30] C. D. Weissman and S. Bobrowski. The design of the force.com multitenant internet application development platform. In *SIGMOD*, pages 889–896, 2009.

[31] P. Xiong, Y. Chi, S. Zhu, H. J. Moon, C. Pu, and H. Hacigümüş. SmartSLA: Cost-Sensitive Management of Virtualized Resources for CPU-Bound Database Services. *Parallel and Distributed Systems, IEEE Transactions on*, 26(5):1441–1451, May 2015.