Predictive Modeling for Management of Database Resources in the Cloud

by

Rebecca Taft

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

Author . . . . . .
Department of Electrical Engineering and Computer Science
May 14, 2015

Certified by . . . . . .
Michael Stonebraker
Professor
Thesis Supervisor

Certified by . . . . . .
Frans Kaashoek
Professor
Thesis Supervisor

Accepted by . . . . . .
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Theses

Predictive Modeling for Management of Database Resources in the Cloud

by

Rebecca Taft

Submitted to the Department of Electrical Engineering and Computer Science
on May 14, 2015, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Public cloud providers who support a Database-as-a-Service offering must efficiently allocate computing resources to each of their customers in order to reduce the total number of servers needed without incurring SLA violations. For example, Microsoft serves more than one million database customers on its Azure SQL Database platform. In order to avoid unnecesssary expense and stay competitive in the cloud market, Microsoft must pack database tenants onto servers as efficiently as possible.

This thesis examines a dataset which contains anonymized customer resource usage statistics from Microsoft's Azure SQL Database service over a three-month period in late 2014. Using this data, this thesis contributes several new algorithms to efficiently pack database tenants onto servers by colocating tenants with compatible usage patterns. An experimental evaluation shows that the placement algorithms, specifically the Scalar Static algorithm and the Dynamic algorithm, are able to pack databases onto half of the machines used in production while incurring fewer SLA violations. The evaluation also shows that with two different cost models these algorithms can save 80% of operational costs compared to the algorithms used in production in late 2014.

Thesis Supervisor: Michael Stonebraker
Title: Professor

Thesis Supervisor: Frans Kaashoek
Title: Professor

Acknowledgments

I would like to thank Willis Lang, Aaron Elmore, Jennie Duggan, Michael Stonebraker, and David DeWitt who have contributed many ideas to this work and much of the text in Chapters 1, 2 and 3. They will all be listed as coauthors on a future conference submission of this work.

I would also like to thank Manasi Vartak and Tucker Taft for helpful discussions on modeling behavior of database replicas.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As many companies and organizations move their services to the cloud, providers of cloud services must learn to manage their resources effectively at scale. Although many cloud providers such as Amazon's EC2 provide application-independent computing resources, many providers are moving to the Platform-as-a-Service (PaaS) model, in which specific applications are hosted and maintained by the provider. In this work, we focus specifically on the Database-as-a-Service model, which Microsoft supports with their Azure SQL Database (ASD) product.

There has been a great deal of research on cloud database systems. Some commercial products (e.g. NuoDB) have focused on "cloud friendly" offerings that support elastic scaling and simple deployment. Much research has focused on supporting *multitenancy* inside a single DBMS instance, such that a single instance hosts multiple customers. Work has ranged from loose coupling through use of virtual machines or running unmodified database engines [25, 7], to very tight coupling, as exemplified by SalesForce.com [24], where all tenants share a common set of RDBMS tables. As discussed in the next chapter, some research examines the problem of tenant packing (or allocation), but the majority of this research focuses on the relatively small scale of hundreds of tenants and tens of servers found in private service offerings.

We instead focus on a much larger class of cloud database service offerings, with hundreds of servers and hundreds of thousands of tenants. In private communication, James Hamilton claimed that Amazon can "stand up" a server for 10% of the cost of smaller

operations. The reasons are extreme economies of scale (deploying servers by the millions), extreme automation of system administration (investing in tools), expert data center construction (building new centers several times a year), and cheap power (locating in the Columbia River valley). Assuming that extreme economies of scale will continue indefinitely and that cloud pricing will eventually reflect costs, then it is clear that very large cloud providers will dominate the landscape. As the cost of public services decreases, it is likely that many large and small enterprises will rely on some form of public cloud.

At this scale a cloud provider will be supporting millions of databases. For example, Microsoft Azure is currently deploying more than one million SQLServer databases on Azure. Obviously, they do not want to deploy these databases, one per node. Hence, a crucial problem is to decide which databases to colocate on specific servers while maintaining performance objectives for each tenant. This problem amounts to solving a multi-dimensional bin packing problem with millions of objects to be binned. At this scale, even minor improvements in packing leads to fewer required servers or fewer SLA violations, which has the potential to save millions of dollars annually. Addressing this problem is the focus of this paper.

Specifically, we evaluate several approaches to tenant packing using resource traces from Microsoft Azure from a three month window in late 2014. Using several predictive models concerning resource usage, we colocate Azure databases with compatible access patterns. We demonstrate that our placement algorithms, particularly the Scalar Static algorithm and Dynamic algorithm described in Chapter 5, are at least a factor of 2 superior to the algorithms that were in use in the data centers we studied in late 2014.

The contributions of this thesis are:

- A comprehensive cost model for a multi-tenant database environment
- Application of this model to a real dataset with hundreds of thousands of databases
- Several scalable algorithms for effectively packing databases onto a lesser number of servers and minimizing the amount of tenant migration required by ASD
- Performance results showing the benefits of our techniques relative to the efficacy of real-world ASD allocations.

In the remainder of this paper we discuss the Azure resource usage data, our preprocessing on this usage data, the predictive models we have built for resource consumption, and the improvement we have seen relative to the existing deployed strategy.

Chapter 2

Background and Related Work

Research on database multitenancy spans a wide variety of problems, ranging from tenant scheduling and placement [20, 9] to the live migration of tenants [14, 11] to the cloud architecture that can facilitate multitenancy [10, 18]. Here, we focus on one set of problems, namely the architectural designs and tenant placement (or consolidation) techniques for a shared-nothing cluster of database servers at massive scale.

2.1 Multitenancy Models

While there are many architectures to enable database multitenancy, three common models have emerged in practice. *Shared Hardware* enables multitenancy through the use of virtual machines (VMs) to isolate tenants from each other. In this model, each server hosts multiple VMs, with each VM hosting a single tenant with its own OS and database process. This model benefits from strong resource isolation, as the VM hypervisor has controls to limit resource consumption for each of its hosted VMs. Additionally, this model provides rich functionality such as live migration and load-balancing. A downside of this model is the number of tenants that can be consolidated on a single machine is limited due to replicated components and uncoordinated resource utilization (e.g., several independent log writers) [7].

An alternative approach is to use a *shared table* model where all of the tenants' data is colocated in the same tables. Within this model several approaches have been proposed to

support flexible schemas [2], however the heap table utilized by Salesforce is the canonical example [24]. This heap/data table stores all tenants' data in a giant table that identifies the tenant Id, table Id, column Id, and cell value as a string. External metadata tables describe the make-up of tables and columns, and pivot tables provide additional functionality, such as indexes and foreign key constraints. This model provides the highest level of consolidation as it has the minimal amount of redundancy and catalog metadata per tenant. The shared table model requires a significant amount of development effort, however, and traditional database interfaces such as SQL require extensive development. Additionally, with tenant data intermixed in the same tables, resource isolation is weak because resource consumption controls must be implemented outside of the database layer.

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. Exactly what components are shared between tenants is implementation-specific. This approach is utilized for its trade-off between resource isolation offered to each tenant and the number of tenants that can be consolidated on a single machine. Approaches with stronger resource controls, such as using a dedicated virtual machine per tenant or dedicated database resources per tenant (e.g., each tenant has a separate log writer), provide better performance guarantees at the cost of a reduced number of tenants that can be consolidated on a single machine.

A shared process commonly employs one of two techniques to manage resource sharing between tenants. (1) Resource allocation uses throttling and reservations to control or limit the physical resources (e.g., disk IOPS) available to each tenant [22]. (2) Soft isolation allows tenants to dynamically share resources without rigid controls [21]. With this approach tenants may become starved for resources, however, if they are colocated with a tenant with an uncomplimentary workload. For example, a server that has too many disk I/O intensive tenants colocated together is likely to experience performance issues. Additionally, the relationship between latency- or throughput-focused service level agreements (SLAs) and allocated resources, like CPU cycles or disk IOPS, is still an open research question. For these reasons, along with ASD's architecture, we focus on tenant placement techniques for a shared process multitenant server utilizing soft isolation.

2.2 Tenant Placement Techniques

Several approaches have been proposed to address tenant placement. Each of these approaches differ in how they model workloads and search for ideal placement. The techniques presented in this paper differ in how they model workloads, in how they evaluate proposed solutions, and in how they scale with increasing numbers of tenants.

Kairos seeks to minimize the number of servers required to host a static number of tenants [8]. Key contributions of Kairos are a technique to probe the active working set in the presence of greedy memory allocation, and a non-linear model of disk I/O when the working set fits into memory. Using this disk model combined with linear CPU and memory models, Kairos identifies a minimal number of servers and a tenant placement plan to maintain transaction throughput as if the tenants were hosted on a dedicated server. Kairos uses an expensive placement strategy that relies on static working sets that fit into the database's buffer pool. Additionally, Kairos is evaluated on a smaller scale than our work; they target a problem with hundreds of tenants on tens of servers, while our work targets a much larger problem. Similar to our work, Kairos is evaluated on real world workloads.

PMAX focuses on tenant placement with cost driven performance objectives [19]. Here, the system is given a service level agreement per tenant, which includes the cost penalties of missed queries. With this information and a server operating cost, PMAX seeks to minimize the total operating cost for a database service. PMAX relies on a dynamic programming algorithm to identify the minimal cost, including the ideal number of servers and expected SLA violations. Additionally, PMAX identifies issues with common bin packing heuristics in a multitenant database system. This technique is evaluated using a single synthetic workload with varying throughput. Our work differs in the varied workloads and placement constraints encountered in a live deployment.

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]. These features include disk I/O, throughput, and operation complexity (i.e. CPU usage) . All tenant workloads are randomly generated from a randomized set of synthetic benchmarks.

Pythia's main goal is to identify the set of tenants that can effectively share a common buffer pool. Similar to Kairos, Pythia is evaluated on a smaller scale than our work, with hundreds of tenants on tens of servers.

Lang et al. present a study that looked to simultaneously address the tenant placement problem along with the server configuration problem [17]. The authors empirically derive "performance characterizing functions" representing each class of hardware server to optimally place tenants from different performance classes and minimize the cost of provisioning a cluster. Their study relied on Integer Linear Programming (ILP) to solve the optimization problem and used different TPC-C throughput objectives as tenant performance objectives. Our study differs from theirs since we no longer rely on synthetic workloads with little to no temporal changes, and instead, we are trying to develop solutions that can handle real-world variability that we see in the Microsoft production traces.

RTP [23] focuses on read-mostly and main-memory workloads, which results in a single load factor considered for each tenant. With a read-mostly workload, RTP allows for tenants to be load-balanced between several servers. The approach also focuses on a robust placement, such that a single node failure would not result in service violations due to a redistribution of workloads from the failed node. Our approach relies on controlled fault domains to ensure that a system remains operable despite node failures. Similar to our work, the workloads are based on real data from SAP. In contrast to our work, RTP relies on read-heavy workloads that allow sharing of query workloads amongst replica sets. We do not make any assumption about the types of workloads running on ASD.

Quasar focuses on general cloud workloads, not database-specific workloads [13]. This approach classifies workloads by scale-out, scale-up, and heterogeneity, and identifies which server configurations will perform best on the class. The approach also models workload interference. A major component of this work is on allocating hardware, not just assigning workloads to servers. Paragon is similar to Quasar, but focuses only on the assignment of workloads and not the allocation of hardware [12]. Paragon uses classification first for workload heterogeneity and interference using collaborative filtering, and then greedily assigns workloads to servers to minimize interference and maximize utilization.

Chapter 3

Microsoft Azure SQL Database

In this paper, we analyze a performance telemetry dataset from Microsoft's Azure SQL Database (ASD) service. An in-depth description of the ASD system [4, 3] is omitted, but in this section we will provide the background necessary for the remainder of this paper. We will also describe the types of information captured by the telemetry dataset, how we processed the data, and the associated challenges along the way.

Microsoft's Azure SQL Database (ASD) service is classified under the industry-standard category of Platform-as-a-Service (PaaS), and 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 19 different regions around the world with varying sizes and varying network and server configurations. In this work, we studied two production clusters from European (*clusterA*) and North American (*clusterB*) regions. The sizes of these two clusters are each between 100 and 150 nodes. On these servers, the ASD service colocates many customer databases together onto each physical server; multiple tenants are housed in one physical SQL Server database in a single process instance, implementing the shared process model (see Chapter 2.)

If all of the databases were identical (i.e., size, workload/utilization, performance requirements, etc.), then finding an efficient allocation of tenants to servers would be fairly straight-forward. Unfortunately, both the performance requirements and the workload/utilization levels of ASD customers are extremely variable (the low cost of storage makes database sizes a secondary concern). In Fig. 3-1, we illustrate this variability with a trace of the log-

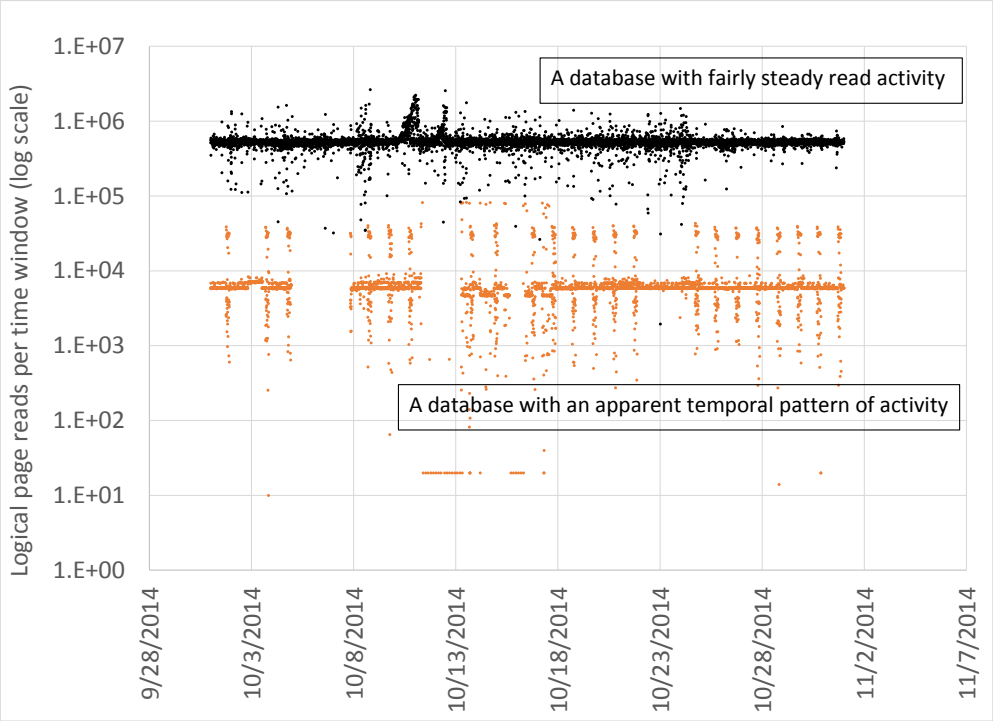


Figure 3-1: Example Resource Utilization by ASD Customer Databases

ical page reads for two customer databases in *clusterB*. This log-scale figure demonstrates how average resource utilization can differ between databases by orders of magnitude, and shows that patterns of resource usage are quite variable both across databases and within a single database over time.

In our data set, we saw a range of ASD customers. ASD customers choose from various service tiers (affectionately called "t-shirt sizes") that correspond to different guarantees of database performance, availability, recoverability and reliability. The original ASD t-shirt sizes were the Web and Business tiers and corresponded only to different maximum database sizes. Databases in these tiers access computing resources through shared pools and have no promised minimum resources. The current subscription model consists of three tiers: Basic, Standard, and Premium. Standard and Premium tiers are further subdivided into different sub-levels. Each of these subscription tiers is differentiated by vary-

ing *database transaction unit* (DTU) performance objectives (a proprietary transactional benchmark metric), as well as maximum DB size, worker threads, session counts, and availability and reliability objectives [3]. In our datasets from *clusterA* and *clusterB*, the vast majority of the customers were still using the original ASD tiers - Web and Business.

ASD places databases using the newer subscription model conservatively such that the minimum amount of resources necessary for the target DTU objective is always available. In other words, the servers are never oversubscribed. The original Web and Business tier customers are more tightly packed in the shared pool since there are no target DTUs. However, there are still conservative limits enforced on the number of these Web and Business tier customers that can be colocated in the shared pool. In this work, we examined the resource consumption of all of the tenant databases (original and current subscription models) in an effort to determine a multitenant placement strategy based on their actual usage patterns instead of the subscription class or conservative fixed caps.

As with all cloud services, ASD is constantly improving and being updated in frequent release cycles. At the time of this study, the ASD service was transitioning through a major version upgrade (V12). Our study is based on the pre-upgrade version since all of the existing customers' databases reside on the pre-upgrade service deployments. The version of the service does not drastically change our approach and conclusions, however, because we are focused on the behavior of ASD customers, not the performance of ASD software.

In the version of ASD running in the two clusters we studied, to satisfy availability objectives, a customer database usually has at least a k-way replication factor with $k = 3$ by default. In ASD, one replica is deemed the "primary" replica and, if $k = 3$, the other two replicas are marked as "secondary" replicas. The k replicas are then stored on separate physical servers. The replicas are asynchronously updated and kept consistent through a quorum commit policy. Secondary replicas may be promoted to primary status through a "swap" operation whereby the old primary then becomes a secondary replica. In this setting only primary replicas directly serve the user's queries. The fact that the primary replicas incur the overwhelming proportion of the resource consumption across the three replicas is important for our experimental setup (Chapter 6).

The ASD service may also physically migrate a customer's database replica from one

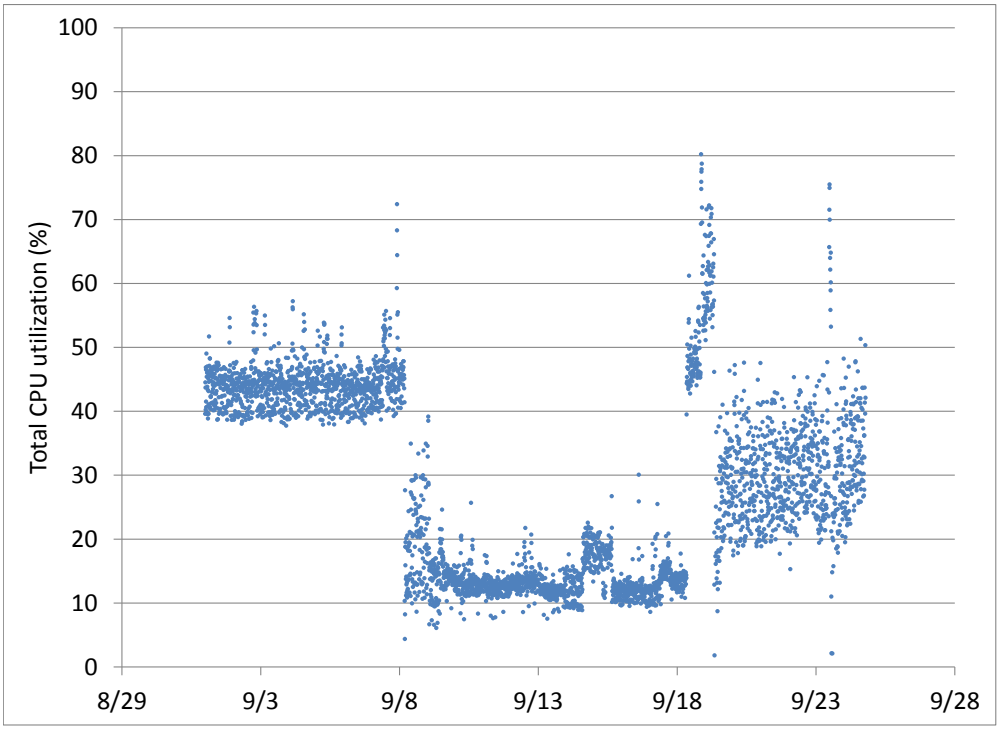


Figure 3-2: Example Server Load Variability

server to another, which can be an expensive operation. Sometimes migration is unavoidable, however, due to server load imbalances. Fig. 3-2 shows the CPU utilization from a server in *clusterB*. We observed that all servers experience some variation in total utilization, but the server depicted in Fig. 3-2 varies dramatically between 80% utilization and 3% utilization within the span of a couple of days.

When replicas are moved (or initially placed), the placement adheres to upgrade and fault domain placement policies. These policies organize groups of servers into any number of domains that make it easier to roll out upgrades and quickly isolate the impact of common failure scenarios.

Attribute	Description
timestamp	The time when this window ends.
interval_seconds	The length 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	A foreign key denoting the machine that this replica resides on during this 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 3.1: Schema of the Microsoft ASD Performance Telemetry Dataset

3.1 Performance Telemetry Dataset

In our work, we studied performance telemetry traces captured over three months in late 2014 from two production ASD clusters: *clusterA* in Europe and *clusterB* in North America. We studied these traces in order to understand and characterize user resource consumption in a cloud setting. For example, by identifying differences between user databases such as the variations in the number of logical page reads (as shown in Fig. 3-1), we hoped to better understand how to consolidate databases together to increase density and multitenancy for greater operational efficiency. Our hypothesis was that training predictive models on these production traces would result in better allocations of databases to machines in comparison to the production assignments used by ASD.

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. For this study, we focused on four resources. The schema of the data is shown in Table 3.1.

Each record in the timeseries represents the resource utilization of a database or its replicas for a time interval. Time intervals are 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; however, due to the variability of the capture pipeline across a production system, this is a best-effort goal. In practice, the length of these recordings varies significantly.

Anonymized customer databases are uniquely identified by the field *dbuid*. The replica type and the server that it resides on are provided since replicas can be swapped or moved as described at the beginning of Chapter 3. The subscription class of the database can be inferred by the *max_worker_count* field as this lists the maximum number of worker threads available for query processing. We found that *max_worker_count* ranges from 0 to 1600 in clusterA and from 0 to 500 in clusterB. In our (2014) datasets, 92% of customers have a *max_worker_count* of 180 since that is the default value for the Web and Business subscriptions.

Finally, the four resources being captured are *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.

3.2 Data Processing

As described in the previous section, each record in the dataset reprents a single replica's activity during a time interval defined by the *timestamp* and *interval_seconds* fields. In an ideal world, 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).

Unfortunately we do not live in an ideal world, so in this study we transform the raw dataset to make each record correspond to a perfectly aligned five-minute window. To perform this transformation, 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 example, a record that starts at 10:03:00 and ends at 10:08:00 is split into two records: one starting at 10:00:00 and ending at 10:05:00, and another starting at 10:05:00 and ending at 10:10:00.

For simplicity, we assume that resource utilization is uniform throughout each record's

timestamp	interval_seconds	cpu_time
10:03:00	300	10000
10:09:00	350	20000

↓

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

↓

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

Figure 3-3: The data pre-processing pipeline. For simplicity we show only three of the ten fields in each record. Assume that all of the above records correspond to the same database replica on the same machine.

duration. Therefore, if a measurement spans multiple windows, each window gets the fraction of the measurement for which it has coverage. In the previous example where we split the record starting at 10:03:00 and ending at 10:08:00, the 10:00:00 window receives 2/5 of the load and the 10:05:00 window is assigned the remaining amount.

This transformation has the potential to double or triple the size of the dataset by splitting each record into two or more reocrds. Therefore, as a final step, we aggregate all rows for each replica for each time window into a single row. See Fig. 3-3 for a simple example of the data pre-processing pipeline.

3.3 Production Challenges

As we gathered the data and prepared the traces for tenant packing, we quickly realized some of the challenges associated with using traces from a production cloud system. In this section, we briefly touch on some of these lessons "from the trenches".

The telemetry data described in Section 3.1 was noisy due to bugs and bug fixes, upgrades, and practical reasons (such as data volume). For example, the telemetry traces began in August 2014, but a bug caused inconsistencies in the measurements for the first half of the month. The bug was fixed at the end of the second week, and we discarded the affected time windows.

Furthermore, as the service evolved, certain (internal and external) subscription classes were created or redefined, making it difficult to determine the service level of some tenants. We worked around this by using the *max_worker_count* as a proxy for the subscription class (see Section 3.1).

Finally, midway through the data collection period, we found that the administrators of the telemetry pipeline had stopped populating the traces of secondary database replicas. The reasons for discarding this data were (1) sheer volume of data (there were roughly twice as many records for the secondary replicas as the primary replicas), (2) the relatively negligible load attributed to the secondary replicas, and (3) the impending rollout of a new back-end architecture that fundamentally changed the replication strategy.

Table 3.2 lists the replica data that we have for each cluster, per month. In October and November, we are lacking the telemetry data for all of the secondary replicas in both clusters. Using the month of September, we calculated summary statistics for the resources listed in Table 3.1. This analysis revealed that the load incurred by the secondary replicas was negligible for all of the metrics except *log operations*. For this field, we found that on average, a single secondary replica incurred 20% of the log operations of a primary replica. Owing to ASD's practice of shipping "log diffs" (page after-images) from the primary to the secondaries, this observation is to be expected. Given that the secondary replicas account for a relatively small portion of the total resource usage, we decided to use only primary replica data for most of our evaluation. See Appendix A for further analysis

Cluster	Month	Primary Data	Secondary Data
clusterA	Sep	X	X
clusterA	Oct	X	
clusterA	Nov	X	
clusterB	Sep	X	X
clusterB	Oct	X	
clusterB	Nov	X	

Table 3.2: Replica Data per Month

of the impact of the missing data on the validity of our results.

Beyond the inconsistencies with the collected data, some data that would have been useful was never collected in the first place. For example, the traces did not directly record when new tenants were created or deleted; we inferred these values conservatively from the traces.

Chapter 4

Problem Definition

We set out to create several different algorithms to pack replicas onto machines, ensuring that no two replicas of the same database are assigned to the same software upgrade unit or fault domain (see Chapter 3).

To evaluate the performance of our algorithms, we used two different baselines:

1. The 2014 Azure allocation, which allows movement of replicas. Referred to as "**Azure**" in the charts.

2. The 2014 Azure allocation, disallowing movement of replicas (i.e., assign each replica to the first machine it appears on for the whole three month period). Referred to as "**Azure no movement**" in the charts.

We define a "violation" as a moment in time when a machine is consuming a particular resource (either logical reads, logical writes, CPU usage, or write-ahead log operations) above some threshold consumption level. See the next section for a more formal definition of what it means for a machine to be in violation. The goal of our algorithms is to incur fewer violations than either of the Azure baselines while also reducing the number of machines used.

Another way to think of this goal is in terms of cost minimization. Each time a machine has a violation, each of the active tenants on that machine may have SLA violations. This may be in terms of availability if the machine is so loaded that requests cannot be fulfilled, or performance if DTUs cannot be delivered (see Chapter 3.) 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 such outages. In this paper, we make the simplifying assumption that our *violations* will require customer compensation. This assumption is similar to the assumption made by [19] that any query processed on an overloaded server will result in an SLA violation. While minimizing service outages is important, 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. Put differently, 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 work is to find a stable packing of database tenants onto machines such that migration of tenants is not frequently necessary. Migration can increase server utilization and has the potential to disrupt the workload of the tenant being migrated, so it is best avoided if possible. In Chapter 5, we introduce algorithms which achieve this goal by moving no tenants at all after the allocation is set (the "static" algorithms), or by moving at most a few tenants per machine per week (the "dynamic" algorithm).

4.1 Metrics for Success

In order to more formally define our metrics for success, we must first define the threshold for what constitutes a violation. From our dataset, we don't know whether a particular tenant would actually experience an 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 very high total utilization of any resource is experiencing a violation.*

So how do we define "very high utilization"? Internally, Microsoft aims to keep total resource usage (e.g., total CPU utilization) of customer databases under 65% of the maximum possible utilization for each machine. We do not know the maximum possible CPU utilization (this is proprietary), so instead we set a lower bound for this value as the maximum total 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. Prior work [15] demonstrated target resource utilization levels set by an administrator can serve as a proxy to control the level of consolidation per machine.

Given that *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 maximum for each resource. That is, we say a machine m containing the set of replicas M_t at time t is in **violation** if the following is true:

$$\begin{aligned} in\_violation_{tm} = & \left(\sum_{r \in M_t} READ_{tr} \right) > P * MAX\_READ \\ & \vee \left(\sum_{r \in M_t} WRITE_{tr} \right) > P * MAX\_WRITE \\ & \vee \left(\sum_{r \in M_t} CPU_{tr} \right) > P * MAX\_CPU \\ & \vee \left(\sum_{r \in M_t} LOG_{tr} \right) > P * MAX\_LOG \end{aligned} \tag{4.1}$$

where $READ_{tr}$, $WRITE_{tr}$, CPU_{tr} and LOG_{tr} represent the logical reads, logical writes, CPU usage, and log operations of tenant r at time t, and P is either 0.65 or 0.85 depending on the chosen threshold. A violation is recorded if any one of these resources exceeds its threshold. *Note that Eq.* (4.1) *is not a definition used internally by Microsoft.* We have created this definition specifically for this study.

Given this definition of a violation, here we define three different metrics for success:

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 hour, using a cost model inspired by Liu et al. [19].

3. **MSFT:** Minimize the total cost of system operation per month, using a cost model inspired by Microsoft Azure's real penalty model [3].

The first metric ("Violations") is self-explanatory: given an alloation of the replicas to y machines, count the number of active tenants assigned to machines experiencing violations during each time slice, and find the average per hour.

The second metric ("PMAX") is an attempt to trade-off the cost of SLA violations with the cost of servers. Similar to the model used in [19], we define the total cost of an allocation with y machines as the cost of y servers plus the cost of each SLA violation during the testing period. As in [19], we use a non-uniform SLA penalty, where each tenant violation is weighted by that customer's *max_worker_count* (See Section 3.1). We also multiply the total SLA penalty by a constant scale factor S in order to give it a proper weight relative to the server cost. Therefore, the cost per hour is defined as:

$$cost = yD + \frac{12S}{n} \sum_{t=1}^{n} \sum_{m=1}^{y} (in\_violation_{tm} * \sum_{r \in M_t} w_r) \qquad \text{(PMAX)}$$

where D is the cost per server per hour, n is the number of time slices in the testing period, $in\_violation_{tm}$ is a binary variable indicating whether or not server m is in violation at time t (See Eq. (4.1)), M_t is the set of active replicas (replicas with a non-zero value for some resource) assigned to machine m at time t, and w_r is the *max_worker_count* of replica r. We multiply the second term by 12 since there are 12 5-minute time slices per hour.

The third metric ("MSFT") is another attempt to trade-off the cost of SLA violations with the cost of servers, 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. (Recall that we are making the *simplifying assumption* that outages will occur when a violation is observed – we are ignoring DTU performance objectives.) Thus, the total cost per month using the MSFT-inspired metric is:

$$cost = yD + \frac{S}{Q} \sum_{q=1}^{Q} \sum_{r \in R} p(v_{rq}) \qquad \text{(MSFT)}$$

given the percentage downtime for replica r in month q is:

$$v_{rq} = \frac{1}{n} \sum_{t=q_1}^{q_n} (in\_violation_{tm} * is\_active_{tr})$$

and the penalty for downtime is:

$$p(v_{rq}) = \begin{cases} 0 & \text{if } 0 \leq v_{rq} < 0.1\% \\ 1 & \text{if } 0.1\% \leq v_{rq} < 1\% \\ 2.5 & \text{if } v_{rq} \geq 1\% \end{cases}$$

where here D is the server cost per month, S is the scale factor for SLA penalties, Q is the number of months in the testing period, R is the set of all database replicas, n is the number of time slices in month q, $in\_violation_{tm}$ is a binary variable indicating whether or not server m is in violation at time t (See Eq. (4.1)), is_active_{tr} is a binary variable indicating whether or not replica r is active (has a non-zero value for some resource) at time t, and m is the machine hosting replica r at time t.

Chapter 5

Tenant Placement Algorithms

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

5.1 Static Allocation

A strawman approach is to randomly allocate databases to machines so each machine gets an equal number of replicas – a **Random** placement algorithm. As we will show in Section 6.2, this approach actually performs well and improves upon the algorithm used by Azure. By simply assigning the same number of randomly chosen replicas to each machine (using a fraction of the total available machines), we can already save at least 60% of Microsoft's operational costs. Here, Microsoft's operational costs are defined as the "cost" for all of the machines we have in the cluster along with the "violation cost", which we derived from [19]. But can we do better? This is the question we attempt to answer in the rest of the paper.

Another approach is to use some portion of the data (e.g., the first two weeks of our dataset) to "train" a model. Then we can use this model to assign replicas to machines and test the performance of our algorithm on the rest of the data. The idea behind this approach is that most databases tend to have predictable workloads with cyclic patterns of resource

usage. If we can classify a tenant based on its historical workload, we can be reasonably confident that this classification will be accurate in the near future. All of the algorithms that follow use this approach.

5.1.1 Predictive Greedy Algorithms

Our first predictive modeling algorithms use a greedy approach in which we assign database replicas to machines one at a time based on two different cost models: a *Scalar* model and a *Summed Time Series* model.

Scalar Cost Model: For our first greedy algorithm, we used a simple scalar cost model based on the average and standard deviation of CPU usage for each database replica during the training period. We found empirically that most replicas in this dataset are CPU-bound and thus CPU is a good predictor for the other three resources, so for simplicity we use only CPU data in our training model. The cost model for testing the allocation still uses all four resources, however (see Eq. (4.1)), since there are a few cases where the resources are not perfectly correlated. Given the CPU training data, we define the "cost" of each replica r during the training period z as:

$$cost_{rz} = base_{rz} + p(w_r, z) \tag{5.1}$$

given base cost:

$$base_{rz} = \frac{1}{n}\sum_{t=z_1}^{z_n}(CPU_{tr}) + \underset{t=z_1}{\overset{z_n}{\sigma}}(CPU_{tr})$$

and padding cost:

$$p(w_r, z) = \begin{cases} \underset{s \in R_z, w_s = w_r}{\text{avg}} (base_{sz}) & \text{if } \exists s \in R_z \text{ s.t. } w_s = w_r \\ \underset{s \in R_z}{\text{avg}} (base_{sz}) & \text{otherwise} \end{cases}$$

where n is the number of time slices in the training period, CPU_{tr} represents the CPU usage

of replica r at time t, R_z is the set of all replicas with activity during the training period, and w_r is the *max_worker_count* of replica r (See Section 3.1). We add some padding $p(w_r, z)$ to each replica's cost in order to account for the fact that some replicas have no telemetry data or very little telemetry data during the training period due to a lack of activity. The padding 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 usage patterns than two unrelated customers.

Assuming that the primary and secondary database replicas have predictable resource usage, the expected value of their CPU usage at any time during the testing period should be equivalent to the average of their CPU usage during the training period. The average does not take into account that some replicas are more predictable than others, however, so we add one standard deviation to the average in order to penalize highly variable replicas. We also tested adding zero and two standard deviations, and found that one standard deviation performed the best.

Using a simple greedy bin packer algorithm, we attempted to evenly distribute the summed cost of the replicas 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. We assume that CPU cost is additive, so the load on a machine is defined to be the sum of the cost of all replicas assigned to it so far. In other words, if we define M as the set of all replicas currently assigned to machine m, the load on machine m during the training period z is:

$$L_{mz} = \sum_{r \in M} (cost_{rz}) \tag{5.2}$$

where $cost_{rz}$ is as defined in Eq. (5.1).

There is an element of randomness in this greedy algorithm because if two replicas have the same cost, either one may be allocated first, resulting in two different allocations of replicas to machines. The number of possible allocations explodes if there are many replicas without training data, since all of these replicas 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 [19, 8]. While a "Best Fit" or ILP approach tends to group the largest replicas together on a few machines, our algorithm ensures that the large replicas are as spread out as possible across all available machines. In Appendix B, we provide background on various bin packer algorithms and justify our approach by showing that on our dataset, grouping the large replicas together creates an allocation that is significantly worse than random.

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

To answer this question, we tried incorporating into our model the time series of resource usage. The reason for using the time series rather than a scalar value is that we prefer to colocate 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 replica on a machine with another tenant that is active at night. This placement is designed to avoid a situation in which multiple tenants on the same machine have a spike in resource usage at the same time.

The second greedy algorithm builds upon the Scalar algorithm described above, but attempts to incorporate the time series of resource usage. Using a similar approach to the Scalar algorithm, it iterates through the replicas in order of decreasing size (as defined by Eq. (5.1)), placing them on machines one at a time. Instead of placing each replica on the currently least loaded machine, however, this algorithm examines how the replica's time series of CPU usage would affect the summed time series on each available machine. Then it chooses the machine on which the maximum value of the summed time series is the smallest.

To avoid overfitting and to make the problem tractable, we aggregated each time series into larger segments (e.g., 6 hours or more) and found the cost from Eq. (5.1) over each segment.

5.1.2 FFT Covariance Cost Model and Algorithm

The next algorithm departs from the greedy model. In this algorithm, we compute the fast Fourier transform (FFT) of the CPU usage time series for each of the highly active "large" primary replicas (the top 5% of replicas based on Eq. (5.1)). To avoid overfitting and to ensure the problem will fit in RAM, we then eliminate all but a fraction of the FFT coefficients for each large primary replica. As an example, since the original time series is at the granularity of five-minute intervals, the first $1/72$nd of the FFT coefficients correspond to wavelengths greater than 6 hours. To smooth out variation under 6 hours, therefore, we truncate an FFT of length 8640 (30 days of data) after 120 values.

In order to determine which replicas to place together on a machine, we perform agglomerative hierarchical clustering with complete linkage. Hierarchical clustering works by iteratively combining items with minimum "distance" into a hierarchy, so here we define the "distance" between replicas 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. Note that we use covariance rather than pairwise correlation since covariance is affected by the magnitude of the FFT coefficients, and in addition to avoiding the colocation of correlated replicas, we also want to avoid the colocation of extremely large (active) replicas.

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 replicas 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 database replicas (the top 5% of replicas based on Eq. (5.1)), as pairwise covariance and clustering of all replicas would be extremely computationally intensive. We allocate the remaining "small" replicas using the Scalar greedy algorithm from Section 5.1.1.

Although this algorithm appears to have great potential, we show in Section 6.2.2 that,

as of this writing, the FFT Covariance algorithm does not improve over the Scalar Static algorithm. Further research is needed to identify the reasons for this result and see if the algorithm can be fixed.

5.2 Dynamic Allocation

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

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 replicas 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 and at risk of causing SLA violations, we can balance the load in one of two ways: (1) by swapping primary replicas on that machine with their corresponding secondary replicas on other under-loaded machines as in [20], and (2) by moving replicas off that machine to other machines with available resources. Option (1) is preferable as swapping is a less expensive operation than moving, but it may not be sufficient to balance the workload. However, the dataset does not include data on secondary replicas in October and November (see Section 3.3), so unfortunately, we could only study option (2) dynamic algorithms.

We created a dynamic algorithm implementing option (2) above, which limits the number of databases that are allowed to move between machines. Starting with a complete allocation of databases to machines and an updated set of training data, we move replicas one at a time from the most overloaded machine to the most underloaded machine (as defined by Eq. (5.2)) until either the load is balanced or we have performed the maximum number of moves allowed. For example, if there are y machines in the cluster and the

Algorithm 1: The FFT Covariance Cost Model Algorithm

Input: time series of CPU usage for each of the x large replicas; cost for each of the large replicas from Eq. (5.1); F size of truncated FFT; list of y available machines

Output: unique assignment of large replicas to machines

Let *max_cost* be $\left(\frac{1}{y}\sum_{i=1}^{x} cost_{r_i}\right) + \varepsilon$;

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

for *i=1* **to** x **do**

  Calculate FFT of the time series for replica r_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 replicas */`

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 the hierarchy to perform hierarchically structured bin packing */`

Set cluster group c_i to $\{\{r_i\}\}$, 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 $distance(c_i, c_j) = \max_{r_a \in c_i, r_b \in c_j} (cov_{a,b})$;

  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;

maximum moves per machine is set to 1, we will perform at most $y/2$ moves total. If the cluster is already perfectly balanced, the algorithm will not move any replicas and will return the same allocation it started with. (Note that we have not changed the cost model from Eq. (5.2); for the dynamic algorithm we make the simplifying assumption that moving replicas is free.)

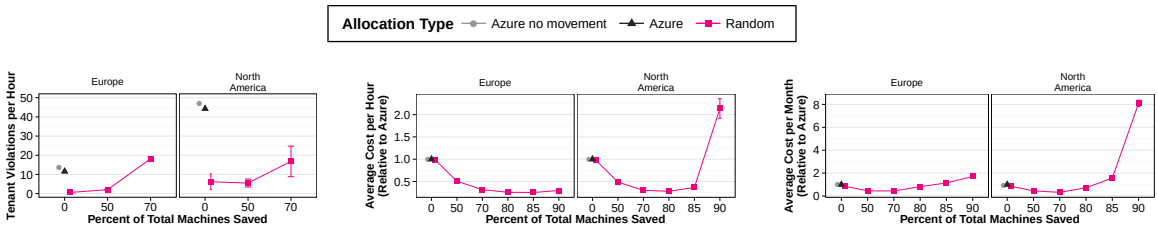
Chapter 6

Evaluation

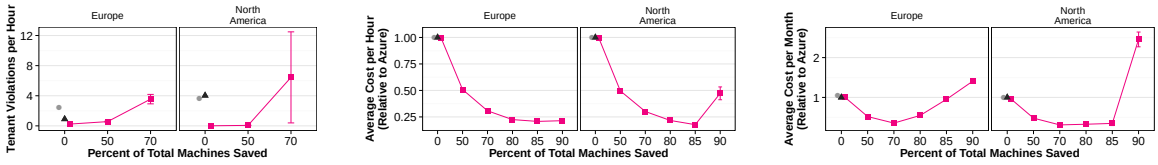
We performed all training and evaluation of our tenant placement algorithms on a Vertica [16] cluster with a mix of SQL queries and user defined functions. Because there is an element of randomness in all of the algorithms (due to the fact that not all replicas are active during the training period; see Section 5.1.1), we ran each agorithm many times in order to evaluate its effectiveness. The metrics used for evaluation are described in Section 4.1.

6.1 Setting Parameters

To calculate the cost of each allocation, we assume that each machine costs 6000 units per hour or 4320000 units per month, based on the default of 100 units per minute in Liu, et al. [19]. The SLA penalty in Liu, et al. is between 2 and 20 units by default, 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). But this is still not quite right, since we are counting 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*). Thus, the PMAX scale factor for SLA penalties in Eq. (PMAX) is set to 3 (*1/10 * 30*). MSFT violations are at the granularity of one month, so each violation in the MSFT model is equivalent to 259200 violations in Liu, et al. (*360 queries per hour * 720 hours per month*). The MSFT cost model does not use *max_worker_count*, so we do not need to divide the



(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

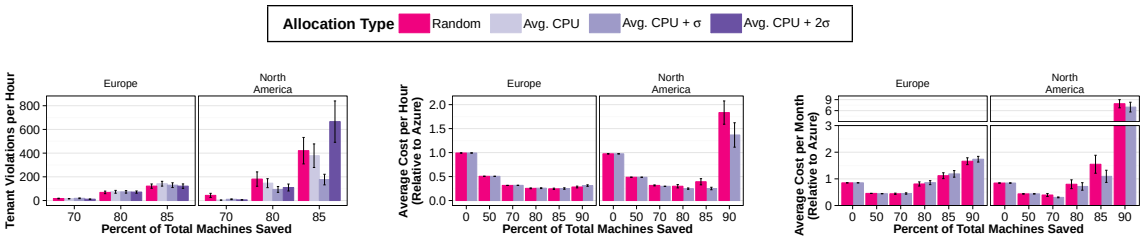
Figure 6-1: Tenant violations and operational cost of the Random allocation scheme and the Azure baselines as a function of total machines saved

SLA penalty by 10, and thus the MSFT scale factor for SLA penalties in Eq. (MSFT) is 259200.

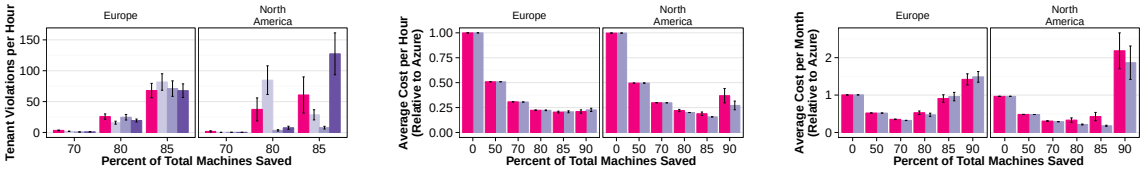
As described in Section 3.3, the ASD dataset did not include data from the secondary replicas in either October or November. Therefore, all the experiments that follow evaluate the allocation algorithms with the primary replicas only. We provide justification for this decision in Appendix A and show that this omission does not invalidate our results.

6.2 Evaluation of Static Algorithms

Fig. 6-1 compares the performance of the Random allocation scheme to the Azure baselines. Fig. 6-1 shows tenant violations and operational cost of the Random allocation scheme and the Azure baselines using the metrics described in Section 4.1, averaged over the entire three-month period (Sep - Nov 2014). In these charts, any machine with resources exceeding 65% (for Figs. 6-1a to 6-1c) or 85% (for Figs. 6-1d to 6-1f) of the maximum is considered to be "in violation" (see Section 4.1). Cost is displayed as a function of the percentage of total machines saved (left unused) for each datacenter. Each data point for the Random allocation scheme represents the average of 10 independent runs of the algorithm.

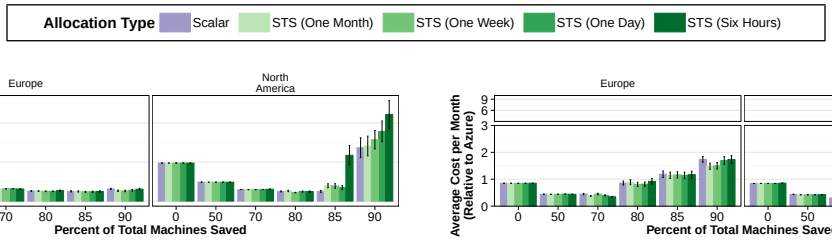


(a) Violations with Threshold at 65% of Max **(b)** PMAX Cost with Threshold at 65% of Max **(c)** MSFT Cost with Threshold at 65% of Max

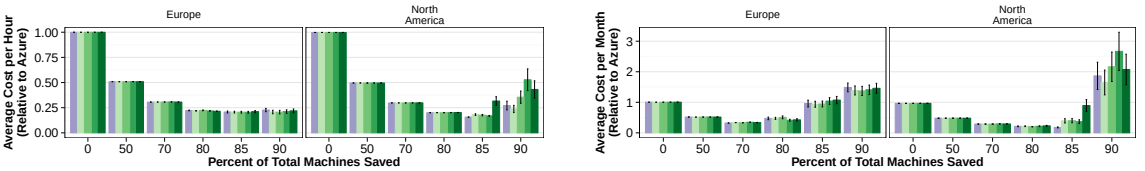


(d) Violations with Threshold at 85% of Max **(e)** PMAX Cost with Threshold at 85% of Max **(f)** MSFT Cost with Threshold at 85% of Max

Figure 6-2: Tenant violations and operational cost of the scalar static allocation schemes compared to random allocation as a function of total machines saved



(a) PMAX Cost with Threshold at 65% of Max **(b)** MSFT Cost with Threshold at 65% of Max

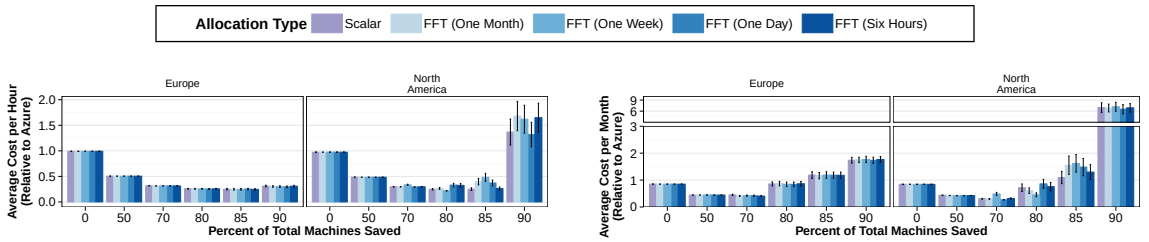


(c) PMAX Cost with Threshold at 85% of Max **(d)** MSFT Cost with Threshold at 85% of Max

Figure 6-3: Operational cost of the Summed Time Series (STS) allocation schemes compared to the scalar allocation scheme as a function of total machines saved

Error bars show the standard error of the mean.

As shown in Figs. 6-1a and 6-1d, a random allocation produces fewer violations than the two Azure baselines when using only half or in some cases even a third of the machines. With only half the machines, the random allocation causes up to 98% fewer violations than



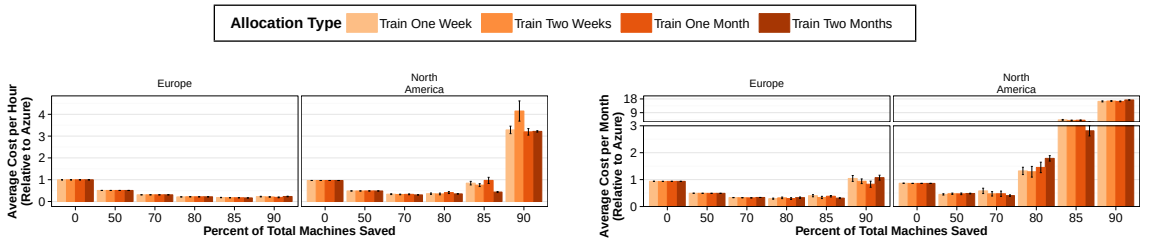
(a) PMAX Cost with Threshold at 65% of Max

(b) MSFT Cost with Threshold at 65% of Max

(c) PMAX Cost with Threshold at 85% of Max

(d) MSFT Cost with Threshold at 85% of Max

Figure 6-4: Operational cost of the FFT allocation schemes compared to the scalar allocation scheme as a function of total machines saved



(a) PMAX Cost with Threshold at 65% of Max

(b) MSFT Cost with Threshold at 65% of Max

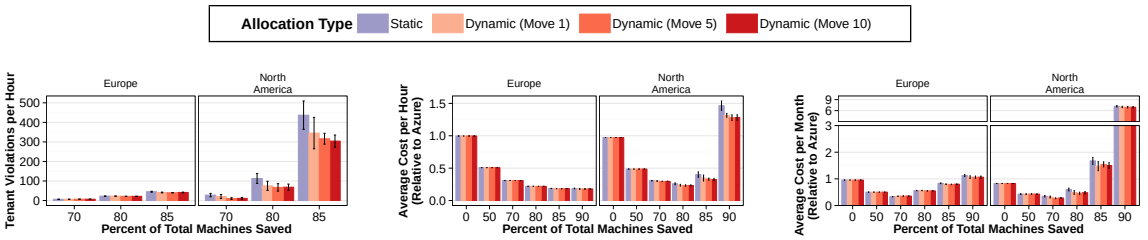
(c) PMAX Cost with Threshold at 85% of Max

(d) MSFT Cost with Threshold at 85% of Max

Figure 6-5: Operational cost of the scalar static allocation with varying training periods as a function of total machines saved

the Azure allocation using all machines.

The other four charts in Fig. 6-1 show the potential cost savings of using a random allocation scheme with fewer machines. By saving 70% to 85% of the server cost, we can save up to 82% of total operational costs with the PMAX model and 69% with the MSFT model. Total violations do increase when we use fewer than 50% of the machines, however, so a more conservative cost model giving higher weight to SLA violations might not attempt to save more than 50% of machines. Still, we achieve a 50% cost savings even



(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 6-6: Tenant violations and operational cost of the dynamic allocation schemes compared to the best static allocation as a function of total machines saved

in the worst case.

6.2.1 Evaluation of Predictive Greedy Algorithms

Scalar Cost Model: To see if using a simple predictive model allowed for a better allocation of databases to machines than random assignment, we compared random allocation to the scalar cost model. In this experiment, we train on two months and test on the third month of the dataset (the random algorithm does not actually use the training data, but we still tested each run on one month for a fair comparison with the scalar cost algorithms). Fig. 6-2 shows the performance of these algorithms, cross-validated across all three months (we switch to a bar chart so it is easier to visualize and compare several tenant-placement algorithms at once). 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. As with Fig. 6-1, any machine with resources exceeding 65% (for Figs. 6-2a to 6-2c) or 85% (for Figs. 6-2d to 6-2f) of the maximum is considered to be "in violation". See Section 4.1 for a detailed description of the different metrics.

The cost model in which we add one standard deviation to the mean (see Eq. (5.1))

consistently performs the best of all four algorithms shown in Figs. 6-2a and 6-2d, so we only show this model compared with Random in the other four charts in Fig. 6-2. When saving 80% of machines in the North American data center at the 85% threshold, the scalar cost model produces 91% fewer violations than a purely random allocation. In the worst case, it is no worse than random, within the margin of error.

It is interesting to note that the scalar cost model shows significant improvement over a random allocation (based on the metrics from Section 4.1) only in the North American data center; the European databases seem to be difficult to predict (All bars for a given percentage of machines saved in Europe are basically the same height). In future work, we hope to classify the behavior of different groups of databases to identify which groups will benefit from predictive modeling and which will not.

Figs. 6-2b, 6-2c, 6-2e and 6-2f also show that the scalar cost model performs better than random when we consider total operational cost. For the 85% threshold in the North American data center, the scalar model costs 17% less than random with the PMAX model and 42% less with the MSFT model (saving 85% of machines). The Azure baselines are not shown in the charts, but the scalar model costs up to 84% less than Azure with the PMAX model and 82% less with the MSFT model. The reason these savings are so high is that the scalar model causes minimal SLA violations while saving up to 85% of the machines used by the baselines.

The static algorithm in which we assign each replica the scalar cost defined in Eq. (5.1) compares favorably to the Azure baselines and the random allocation, as well as to the variations in which we add zero or two standard deviations to the average rather than one. Based on these results, it seems that one standard deviation acheives the right blanace between being close to the expected value and penalizing highly variable replicas.

Summed Time Series Cost Model: To see if there is any benefit to using a time series instead of a scalar value for the cost of each replica, we ran our Summed Time Series algorithm (see Section 5.1.1) with four different segment lengths: one month, one week, one day, and six hours. 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. 6-3 shows the performance of these algorithms. As in Fig. 6-2, each data point represents the average of 30 independent runs of the algorithm; 10 runs for each of the three months.

Surprisingly, using a time series does not produce better results than using a scalar value for each replica. In the case of the one-day and six-hour segments, it seems that the time series actually produces significantly worse results than the scalar value. Presumably the poor performance is due to over-fitting; our model assumes that database replicas have cyclic resource usage when that may not actually be the case.

In future work, we hope to investigate how to better use the time series model. Perhaps we need to first classify the database replicas into those that exhibit periodicity and those that do not. We are confident that this time series model will be useful for some datasets, but it seems that it is not the correct model for the two Azure clusters in this study.

6.2.2 Evaluation of the FFT Covariance Model

We tested our FFT Covariance model (see Section 5.1.2) with the first 3, 10, 62 and 245 FFT coefficients corresponding to minimum wavelengths of one month, one week, one day and six hours, respectively. As with our evaluation of the scalar cost model and time series model, we trained our models on two months of data and tested on the third month, cross-validating across all three months.

As with the Summed Time Series model, the FFT Covariance model does not show any significant improvement over the simple scalar model on this dataset. More research is needed to determine if this model will be successful on other datasets.

6.3 Motivation for Dynamic Algorithm and Length of Training Period

As we show in the previous section, a simple scalar model is the best way to create a static allocation of database replicas to machines. But what if we allow movement of replicas? Can we do better than a static allocation?

To motivate the need for a dynamic algorithm, we ran some experiments in which we varied the amount of training data available to the static algorithm. Specifically, we ran experiments in which we trained on one week (September 1 - 7), two weeks (September 1 - 14), one month (September 1 - 30), and two months (September 1 - October 31), and tested on the month of November. The results of these experiments are shown in Fig. 6-5. Each bar in the charts represents the average of 10 runs of the algorithm, and error bars show the standard error of the mean.

Although the trend is not perfect, it's clear that a longer training period tends to produce better results. For example, training on two months almost always produces a lower-cost allocation than training on one month or less. Given these results, we move on to evaluation of our dynamic algorithm.

6.4 Evaluation of Dynamic Algorithm

Starting with our best static allocation, 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 replicas between machines each time.

For the baseline static algorithm, we trained the scalar model (see Eq. (5.1)) 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 three variations of the dynamic model. In the first variation, we ran the dynamic algorithm once for each week in the dataset. Each time, we started with the previous week's allocation and moved at most one replica onto or off of each machine. Then we tested the performance of the new allocation on the following week.

The second and third variations were almost the same as the first, but allowed five or ten replicas to move onto or off of each machine during each cycle instead of only one.

Fig. 6-6 shows the performance of the dynamic algorithms compared to the static algorithm. Each bar is the result of 10 runs of the algorithm. The different metrics are as described in previous sections. In all cases, the dynamic algorithms perform at least as well

as the static algorithm, and in most cases outperform the static algorithm. When saving 80% of machines in the North American data center at the 85% threshold, the dynamic algorithm allowing 10 moves per cycle causes 88% fewer violations than the static algorithm. Additionally, the dynamic algorithm with 10 moves per cycle costs 21% less than the static algorithm using the PMAX model, and 25% less using the MSFT model.

Chapter 7

Conclusion

This thesis studied a large dataset from Microsoft's multitenant database system, Azure SQL Database. We used this dataset to compare the effectiveness of several different tenant placement algorithms, including the algorithm used by Microsoft in production at the time the data was collected. We introduced two cost models which take into account server costs and SLA penalties, and used these models to evaluate the tenant placement algorithms. We showed that with these cost models, a random assignment of the databases to a subset of the machines used in production has the potential to save up to 82% of total operational cost by saving up to 85% of the machines used in production. Furthermore, we introduced a simple cost model using CPU usage data and showed that it can improve upon the random allocation by up to 42%. Finally, we described a dynamic version of this algorithm which allows periodic tenant migration, and showed that it can improve on the static algorithm by up to 25%.

We also introduced a couple of more complex algorithms which take the time series of CPU usage into account and attempt to co-locate databases with anti-correlated usage patterns. We showed that in their current form these algorithms do not improve upon the simple cost model, but in future work we hope to refine these models and classify tenants into those that will benefit from more complex models and those that will not.

Appendix A

Allocation of Primary Replicas Only

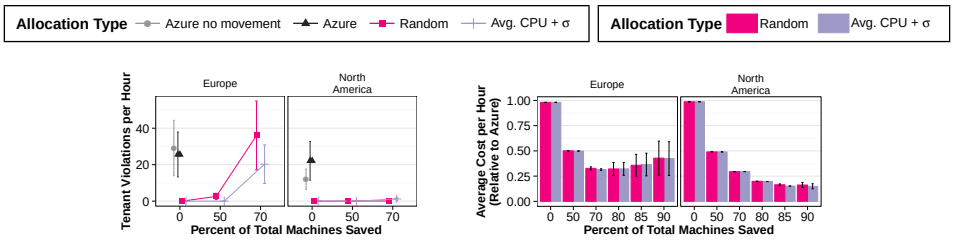
With the secondary replica omission from October and November, we devised two sets of experiments to show that training and testing with data from only primary replicas is a valid approach and produces similar results to training and testing on all replicas:

- Using only the primary replica data in September, we train our models on one week and evaluate our approach using the remaining weeks in September.
- Using all of the available data in September, we train our models on one week and evaluate our approach using the remaining weeks in September.

Figs. A-1 and A-2 show the results of these experiments. Each data point in these charts represents the average of 12 independent runs of the algorithm; three runs for each of the four weeks in September. Error bars show the standard error of the mean.

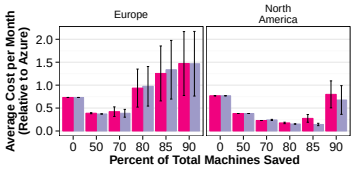
The first thing to notice is that the trends are very similar between the two sets of charts. In all cases, the Random and Avg. CPU + σ (scalar static) allocations cause fewer violations than the Azure baselines when using all or half of the machines. When using only 30% of machines, both sets of charts show that our allocations always cause fewer violations than the Azure baselines in the North American data center and sometimes in the European data center. Both sets of charts also show that the scalar model always performs at least as well as Random (within the margin of error), and often better than Random.

Not surprisingly, the main difference between the two sets of charts is that including secondary replicas in the analysis results in more violations and thus a higher cost. While

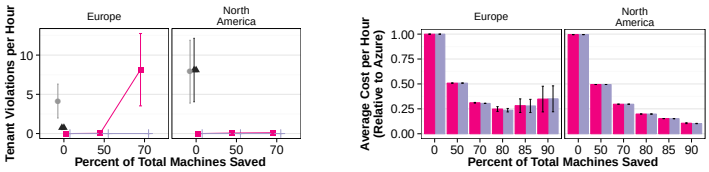


(a) Violations with Threshold at 65% of Max

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

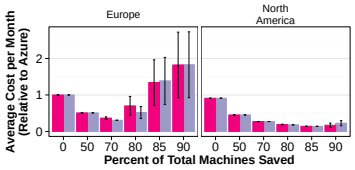


(c) MSFT Cost with Threshold at 65% of Max



(d) Violations with Threshold at 85% of Max

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

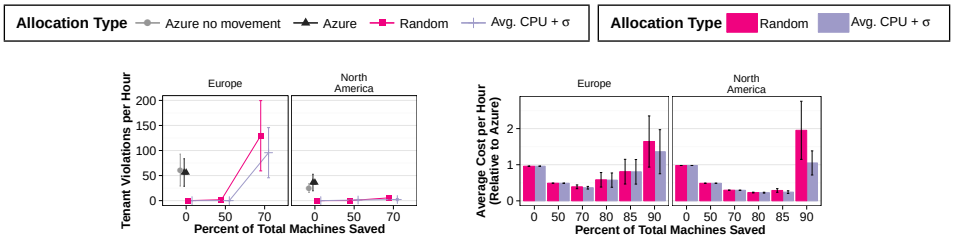


(f) MSFT Cost with Threshold at 85% of Max

Figure A-1: Tenant violations and operational cost of the Random allocation scheme, the Scalar Static allocation scheme and the Azure baselines as a function of total machines saved, using **primary replicas only**

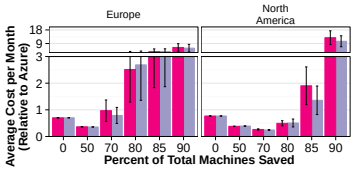
the optimal number of machines saved with only primary replicas is between 50 and 80% for Europe and between 85 and 90% for North America, with secondary replicas included, the optimal value shifts to between 50 and 70% for Europe, and between 70 and 85% for North America.

Although the exact numbers change with secondary replicas included, the takeaway is the same: we can save a significant amount of operational costs by saving at least 50% of the machines used by Azure.

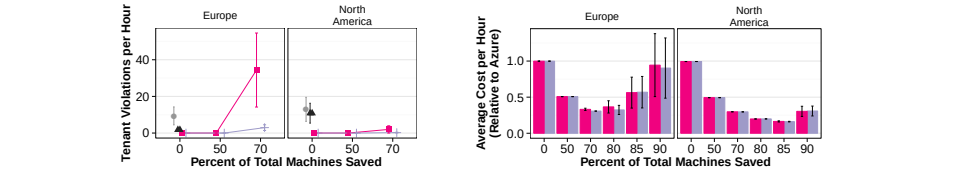


(a) Violations with Threshold at 65% of Max

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

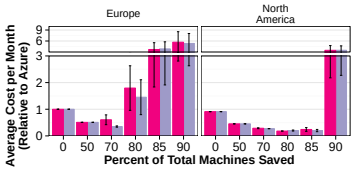


(c) MSFT Cost with Threshold at 65% of Max



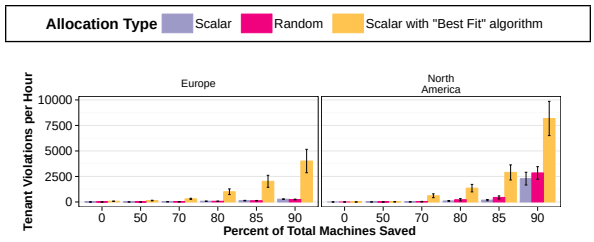
(d) Violations with Threshold at 85% of Max

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

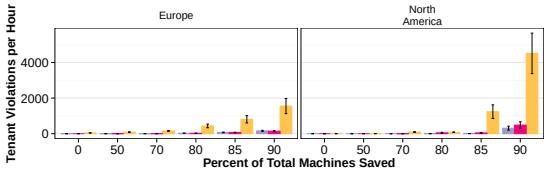


(f) MSFT Cost with Threshold at 85% of Max

Figure A-2: Tenant violations and operational cost of the Random allocation scheme, the Scalar Static allocation scheme and the Azure baselines as a function of total machines saved, using **both primary and secondary replicas**



(a) Violations with Threshold at 65% of Max



(b) Violations with Threshold at 85% of Max

Figure B-1: Tenant violations of the Scalar allocation scheme using a "Best Fit" greedy algorithm, compared to the Random allocation scheme and the regular Scalar allocation scheme as a function of total machines saved

Appendix B

Choice of Bin Packing Algorithm

As described briefly in Section 5.1.1, we use a greedy algorithm for tenant placement which sorts replicas by decreasing size and iteratively places each replica on the **least loaded** machine, until all replicas have been placed. As described in [1, 6], this algorithm is (unfortunatly) commonly known as the "Worst Fit" greedy algorithm. The algorithm is referred to as "Worst Fit" since it can lead to an inefficient use of space when packing items of known size. It is important to note, however, that in our case we do not have perfect knowledge of the "size" of each replica. We estimate the size based on the training data, but we cannot

perfectly predict performance during the testing period. To show the benefit of our approach, we compare the performance of the algorithm from Section 5.1.1 to an algorithm using the same cost model, but a "Best Fit" rather than "Worst Fit" approach to packing (see [1, 6]). The "Best Fit" algorithm sorts replicas by decreasing size and iteratively places each replica on the **most loaded** machine that it fits on. We set the capacity of each machine to the average load across all machines + 1%.

We compared the "Best Fit" algorithm to the Scalar algorithm from Section 5.1.1 and the Random allocation by training on two months and testing on the third month of the dataset (the random algorithm does not actually use the training data, but we still tested each run on one month for a fair comparison with the other two algorithms). Fig. B-1 shows the performance of these algorithms, cross-validated across all three months. Each data point represents the average of 15 independent runs of the algorithm; 5 runs for each of the three months. Error bars show the standard error of the mean.

Clearly, the "Best Fit" approach performs terribly. By placing all of the most active replicas on the same machine, it creates a pathological allocation that is significantly worse than random. In contrast, the regular Scalar approach using the "Worst Fit" algorithm ensures that the large replicas are as spaced out as possible. This result indicates that our cost model is not perfect - there is some extra hidden cost in the large replicas that we do not account for - but it's difficult to see what a better model would be. Luckily, we can overcome shortcomings in the cost model by using the "Wort Fit" bin packing algorithm.

Another popular approach to the bin packing problem is to use an Integer Linear Programming (ILP) algorithm to find an optimal packing of the tenants given a set of constraints. For the same reason that the "Best Fit" approach failed, we do not expect the ILP approach to perform well on our problem, since it also attempts to distribute the cost as evenly as possible without keeping large replicas separated. We could potentially add constraints to limit the number of small, medium, and large replicas allowed on each machine, but it seems unlikely that it would perform better than our algorithm. Furthermore, at the scale we are working, this becomes an extremely large and complex linear program. In order to make the problem tractable, we would need to constrain the problem by grouping together the smaller replicas and assigning them in chunks. Even then, the progam would

likely take an unacceptably long amount of time to run for practical applications. For all of these reasons, we chose to focus on simple greedy algorithms.

Appendix C

Failed Ideas

In addition to the ideas presented in Chapter 5, we investigated several other ideas that proved to be less effective.

C.1 Linear Combination Cost Model

For the Linear Combination Cost Model, we define the "cost" of each replica to be the average plus one standard deviation of a linear combination of the four resources. Here, the cost of each replica r during training period z is defined to be:

$$cost_{rz} = \frac{1}{n}\sum_{t=z_1}^{z_n}(COMB_{tr}) + \overset{z_n}{\underset{t=z_1}{\sigma}}(COMB_{tr}) \tag{C.1}$$

where n is the number of time slices in the training period, and $COMB_{tr}$ is defined as:

$$COMB_{tr} = READ_{tr}/MAX\_READ + WRITE_{tr}/MAX\_WRITE + CPU_{tr}/MAX\_CPU + LOG_{tr}/MAX\_LOG \tag{C.2}$$

where $READ_{tr}$, $WRITE_{tr}$, CPU_{tr} and LOG_{tr} represent the logical reads, logical writes, CPU usage, and log operations of replica r at time t, and MAX_READ, MAX_WRITE, MAX_CPU and MAX_LOG are defined as in Section 4.1.

Using the same greedy bin packer algorithm as in Section 5.1.1, we attempted to evenly distribute load across the machines. For simplicity, we again assume that cost is additive, and thus Eq. (5.2) still applies.

This approach turned out to be ineffective and produced results that were no better than a random allocation.

C.2 Four-Dimensional Cost Model

For the Four-Dimensional Cost Model, we use a four-dimensional cost, where the four dimensions correspond to the maximum usage of each of the four resources during the training period. Formally, the 4-d cost of each replica r during the training period z is:

$$cost_{rz} = \langle \max_{t=z_1}^{z_n}(READ_{tr}), \max_{t=z_1}^{z_n}(WRITE_{tr}), \max_{t=z_1}^{z_n}(CPU_{tr}), \max_{t=z_1}^{z_n}(LOG_{tr}) \rangle \qquad \text{(C.3)}$$

where the variables are defined as in Appendix C.1.

As with the previous algorithm, this algorithm uses a greedy approach to packing in which it repeatedly places the largest (highest cost) database replica not yet allocated onto the current emptiest (least loaded) machine. Because this cost model has four dimensions, we rotate through the dimensions as we allocate replicas. For example, if we start by allocating the replica with highest logical read cost, next we would allocate the database replica with the highest logical write cost, then highest CPU cost, and so on. In each case, we try to place the replica on the least loaded machine for the current cost dimension, unless doing so would cause that machine to exceed its capacity along one of the other dimensions. Initially, we set the capacity to the average load across all machines + 1%. If it becomes impossible to place a replica without exceeding any machine's capacity, then we increase the capacity incrementally by 1% each time until placement is feasible.

As with the Linear Combination Cost Model, this approach turned out to be ineffective. It seems that optimizing for more than one resource at a time is very difficult.

Bibliography

[1] Bin packing. http://www.cs.arizona.edu/icon/oddsends/bpack/bpack.htm, 2015.

[2] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *SIGMOD*, pages 1195–1206, 2008.

[3] Microsoft corporation. http://azure.microsoft.com/en-us/documentation/services/sql-database/, 2015.

[4] P.A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D.B. Lomet, R. Manne, L. Novik, and T. Talius. Adapting microsoft sql server for cloud computing. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1255–1263, April 2011.

[5] Bruno Codenotti, Gianluca De Marco, Mauro Leoncini, Manuela Montangero, and Massimo Santini. Approximation algorithms for a hierarchically structured bin packing problem. *Inf. Process. Lett.*, 89(5):215–221, 2004.

[6] Jr. Coffman, E.G., M.R. Garey, and D.S. Johnson. Approximation algorithms for bin-packing - an updated survey. In G. Ausiello, M. Lucertini, and P. Serafini, editors, *Algorithm Design for Computer System Design*, volume 284 of *International Centre for Mechanical Sciences*, pages 49–106. Springer Vienna, 1984.

[7] Carlo Curino, Evan Jones, Raluca Popa, Nirmesh Malviya, Eugene Wu, Sam Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational Cloud: A Database Service for the Cloud. In *CIDR*, pages 235–240, 2011.

[8] Carlo Curino, Evan P.C. Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 313–324, New York, NY, USA, 2011. ACM.

[9] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010.

[10] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems (TODS)*, 38(1):5, 2013.

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

[12] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM.

[13] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM.

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

[15] Aaron J. Elmore, Sudipto Das, Alexander Pucher, Divyakant Agrawal, Amr El Abbadi, and Xifeng 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] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *Proc. VLDB Endow.*, 5(12):1790–1801, August 2012.

[17] Willis Lang, Srinath Shankar, Jignesh Patel, and Ajay Kalhan. Towards multi-tenant performance slos. In *ICDE*, pages 702–713, 2012.

[18] Rui Liu, Ashraf Aboulnaga, and Kenneth Salem. Dax: a widely distributed multitenant storage service for dbms hosting. *PVLDB*, 6(4):253–264, 2013.

[19] Ziyang Liu, Hakan Hacigümüş, Hyun Jin Moon, Yun Chi, and Wang-Pin Hsiung. Pmax: Tenant placement in multitenant databases for profit maximization. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 442–453, New York, NY, USA, 2013. ACM.

[20] Hyun Jin Moon, Hakan Hacigümüs, Yun Chi, and Wang-Pin Hsiung. Swat: a lightweight load balancing method for multitenant databases. In *EDBT*, pages 65–76, 2013.

[21] Barzan Mozafari, Carlo Curino, and Samuel Madden. Dbseer: Resource and performance prediction for building a next generation database cloud. In *CIDR*, 2013.

[22] Vivek R Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. Sqlvm: Performance isolation in multi-tenant relational database-as-a-service. In *CIDR*, 2013.

[23] Jan Schaffner, Tim Januschowski, Megan Kercher, Tim Kraska, Hasso Plattner, Michael J. Franklin, and Dean Jacobs. Rtp: Robust tenant placement for elastic in-memory database clusters. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 773–784, New York, NY, USA, 2013. ACM.

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

[25] Pengcheng Xiong, Yun Chi, Shenghuo Zhu, Hyun Jin Moon, C. Pu, and H. Hacgumus. 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.