Automated Demand-driven Resource Scaling in Relational Database-as-a-Service

Sudipto Das Feng Li Vivek R. Narasayya Arnd Christian König

Microsoft Research
Redmond, WA 98052, USA
{sudiptod, fenl, viveknar, chrisko}@microsoft.com

ABSTRACT

Relational Database-as-a-Service (DaaS) platforms today support the abstraction of a *resource container* that guarantees a fixed amount of resources. Tenants are responsible for selecting a container size suitable for their workloads, which they can change to leverage the cloud's elasticity. However, automating this task is daunting for most tenants since estimating resource demands for arbitrary SQL workloads in an RDBMS is complex and challenging. In addition, workloads and resource requirements can vary significantly within minutes to hours, and container sizes vary by orders of magnitude both in the amount of resources as well as monetary cost. We present a solution to enable a DaaS to auto-scale container sizes on behalf of its tenants. Approaches to auto-scale stateless services, such as web servers, that rely on historical resource *utilization* as the primary signal, often perform poorly for stateful database servers which are significantly more complex. Our solution derives a set of robust signals from database engine telemetry and combines them to significantly improve accuracy of demand estimation for database workloads resulting in more accurate scaling decisions. Our solution raises the abstraction by allowing tenants to reason about monetary budget and query latency rather than resources. We prototyped our approach in Microsoft Azure SQL Database and ran extensive experiments using workloads with realistic time-varying resource demand patterns obtained from production traces. Compared to an approach that uses only resource utilization to estimate demand, our approach results in $1.5\times$ to $3\times$ lower monetary costs while achieving comparable query latencies.

Keywords

Relational database-as-a-service; elasticity; auto-scaling; resource demand estimation.

1. INTRODUCTION

Relational Database-as-a-Service (*DaaS*) is a rapidly growing business with offerings from major providers, such as Amazon's Relational Database Service (RDS), Microsoft's Azure SQL Database (Azure SQL DB), and Google's Cloud SQL. Many enterprises today deploy mission-critical databases in these DaaS environments.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3531-7/16/06...$15.00

DOI: http://dx.doi.org/10.1145/2882903.2903733

Since these DaaS environments are multi-tenant, performance predictability through resource isolation has been identified as a critical requirement for such mission-critical databases [10, 15]. Commercial DaaS offerings now support resource isolation through logical or physical **containers**. The container can be a virtual machine (VM) [10, 20] dedicated to a tenant's database, such as in Amazon RDS, or a logical container, such as in Azure SQL Database [15]. Regardless of the specific container abstraction, a container guarantees a fixed set of resources, referred to as the **container size**.

A major attraction of DaaS is the promise of elasticity, pay-per-use, and high availability. DaaS platforms offer a collection of container sizes and supports elasticity by allowing a tenant to change the container size for their database over time.[1,2,3] Such container size changes can be performed relatively quickly (e.g., typically within minutes). Judiciously changing container sizes results in significant cost savings due to two reasons. First, the workload's resource requirements change over time. Our analysis of resource utilization from thousands of production tenant databases (details in Section 2.2) reveals that changes in resource requirements crossing container size boundaries are frequent, typically occurring in the order of minutes to a few hours. This is similar to resource demand variations observed in non-database workloads [8, 19]. Second, there are large differences in resources and cost between different container sizes. For example, at the time of writing, three orders of magnitude separate the cost of the smallest and largest containers in Azure SQL DB.

We focus on the "scale-up" form of auto-scaling, where the size of a *single* container is changed over time. This is different from the "scale-out" approach of auto-scaling [4,8,19,20] which changes the *number* of containers (e.g., VMs). The "scale-out" approach repartitions the workload as the number of containers change. This orthogonal "scale-up" approach is supported in today's DaaS and allows databases contained in a single server to leverage elasticity without partitioning the database. Even though the maximum container size is limited by the largest server in the service, based on current trends, large containers with $16 - 32$ cores, hundreds of Gigabytes of RAM and Terabytes of storage are sufficient for an overwhelming majority of tenants.

Today, to benefit from elasticity, a tenant has to determine when to scale the container size. Well-known application-agnostic approaches rely on resource utilization as the primary driver for scaling decisions. For instance, if resource utilization is high (e.g., 80% I/O utilization in the current container with 100 IOPS), then scale up the container; similarly, low resource utilization implies scale down. However, high resource utilization does not necessarily

[1]http://azure.microsoft.com/en-us/pricing/details/sql-database/
[2]https://developers.google.com/cloud-sql/pricing
[3]http://aws.amazon.com/rds/pricing/

mean that there is **demand** for more resources. That is, if allocated a larger container (e.g., with 200 IOPS), the workload will indeed consume the additional resources (i.e., number of I/Os will exceed 100 IOPS). *Since resource demand cannot be measured*, the problem is to *estimate* demand for database workloads. The challenge arises due to the complexity of database engines and how multiple resources interact. For example, if the offered load increases, it does not necessarily mean that that adding more resources will significantly improve query latencies, particularly if queries are mostly waiting for locks on shared data items. Similarly, adding more memory might reduce the need for I/O and increase the CPU demand since more data can be cached. When container sizes vary significantly in resources and cost, the penalty for incorrect demand estimation can be high – either poor performance if demand is underestimated or higher monetary cost if demand is overestimated. Most tenants of a DaaS cannot afford to hire sophisticated database administrators necessary to make judiciously scale resources.

We study the problem of how a DaaS platform can support robust auto-scaling functionality on the tenant's behalf by estimating resource demands for the tenant's workload. Any solution that the DaaS platform uses must be general enough for arbitrary, and often unseen, SQL workloads that the tenant's application may issue. Previous work on resource modeling in classic enterprise consolidation scenarios build resource models assuming the workload is known in advance [3, 14, 21] (e.g., the query classes and their relative frequencies are known). However, obtaining such a "representative" workload to train models specific to the workload is hard even with expert database administrators, and is infeasible for a DaaS platform which must support hundreds of thousands of tenants with very different requirements. Therefore, our auto-scaling solution leverages generic telemetry and execution characteristics of a tenant's workload that is available for all tenants and does not require tenant-specific human input.

Auto-scaling functionality supported today in commercial cloud platforms uses tenant-specified rules based on resource utilization.[4,5] We show that for database workloads, it is possible to significantly improve accuracy of demand estimation by deriving additional signals from database engine telemetry that goes well beyond resource utilization. For example, we show that *resource waits*, which is the amount of time a tenant's requests wait for a logical or physical resource, is an important signal since significant waits for resources might indicate the workload would benefit from more resources.

Our technique results in improved accuracy in demand estimation due to three key ideas. First, we derive statistically-robust signals that can tolerate noise, which is inevitable in system telemetry (Section 3). Second, we use domain knowledge of database engine internals to systematically design a decision logic to combine multiple signals to greatly reduce the fraction of inaccurate estimations of high (or low) demand (Section 4). Intuitively, if there are multiple weak signals of high demand for a resource (such as utilization and waits), it increases the likelihood of the demand actually being high. Third, we leverage the fact that a DaaS platform can observe telemetry of large numbers of tenants with very different workloads and resource demands. We analyze this service-wide telemetry to improve our demand-estimation logic, e.g., to determine meaningful and suitable thresholds for the input signals.

Besides estimating resource demands, an end-to-end auto-scaling solution must cater to a number of practical challenges. First, many tenants have a cost budget which is specified over a longer period of time such as a month, while container sizing actions occur more frequently, e.g., in minutes or hours. The challenge is to design an online budget allocation strategy that allows periods of high resource demand (where the budget is consumed at a rate higher that the average rate) while ensuring that the total cost does not exceed the budget. We borrow ideas from the traffic shaping problem in computer networks to address this problem (Section 5). Second, tenants may wish to specify latency goals for their applications. We use this input to further reduce costs. If latency goals are met, we allocate a smaller container even if there is demand for a larger container. Further, latency goals might not be met due to issues beyond resources, such as poorly-written application code. Therefore, we increase the container sizes *only* if there is resource demand, even when the latency goals are not being met. Note that tenants do not need to specify a throughput goal. When the offered load increases, the unmet resource demand and query latencies provide the necessary feedback to the auto-scaling logic. Our solution continuously monitors the resource demands and latency goals, and adjusts the container sizes in a closed loop. We present an end-to-end auto-scaling solution for relational DaaS that *helps the tenants achieve their desired latency goals while minimizing the costs and remaining within the specified budget* (Section 6). Such functionality raises the abstraction of a DaaS by allowing tenants to reason about budget and query latency rather than resource provisioning.

We prototype our solution in Microsoft Azure SQL Database and demonstrate the benefits using a variety of benchmark workloads and resource demand traces derived from production workloads. When compared to an approach that uses only resource utilization to estimate demand, our solution reduces the tenant's costs by $1.5\times$ to $3\times$ while resulting in similar 95^{th} percentile latencies.

Following are the major contributions of this paper:

- By leveraging domain knowledge of database engines, we identify a set of statistically-robust signals and propose a decision logic to improve accuracy in estimating resource demands.
- We propose an end-to-end solution for auto-scaling in a DaaS that combines automatic estimated resource demands with tenant-specified budget constraints and latency goals.
- We demonstrate the cost-effectiveness and robustness of our approach by prototyping it in Microsoft Azure SQL Database and evaluation using a variety of benchmark workloads and resource demand patterns derived from production traces.

2. PROBLEM SETTING

2.1 Database-as-a-Service

We consider a relational DaaS such as Azure SQL Database. The DaaS offers a set of resource containers each with a fixed set of resources (e.g., two virtual cores, 4GB memory, 100 disk IOPS, and 1 TB disk space) and a cost per **billing interval** (e.g., 50 cents/hour).

We consider a setting where container sizes can be scaled independently in each resource dimension or in lock-step for all resources. Figure 1 demonstrates this scaling of container sizes using two resource dimensions CPU and Disk I/O as examples. Container sizes $(S, M, L, \ldots)$ scale both the CPU and Disk I/O in the same proportion. That is, M has $2\times$ larger allocation and L has $3\times$ larger allocation for both CPU and Disk IO compared to S. $S, M_C, L_C, \ldots$

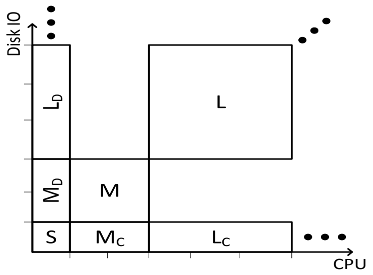
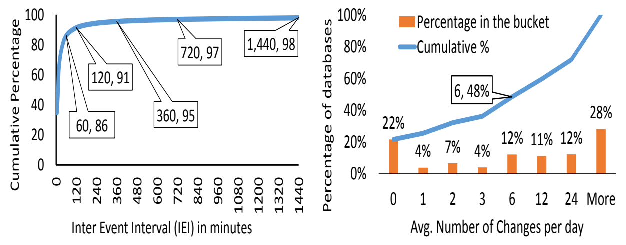


Figure 1: Container sizes in a DaaS with resources as dimensions.

[4]http://aws.amazon.com/autoscaling/

[5]https://cloud.google.com/compute/docs/autoscaler/



(a) CDF of IEI in minutes. (b) Change frequency distribution.

Figure 2: Distribution of interval between changes (IEI) and the frequency of changes.

and $S, M_D, L_D, \ldots$ represent container scaling along CPU and Disk IO dimensions respectively. For instance, the standard tiers in Amazon and Azure proportionally increase resources while high memory, high CPU, and high I/O instances scale resources along a specific dimension. Workloads having demand in one resource can benefit if containers are scaled independently in each dimension.

2.2 Resource Demand Analysis in Production

To understand the importance of auto-scaling, we performed an offline analysis using week-long resource utilization traces for a few thousand tenant databases from production Azure DB clusters. We aggregated a tenant's resource utilization over 5 minute intervals for CPU, I/O, and memory. We then logically *assigned* the smallest container supported by the service that can meet the resource requirements for that interval as the tenant's container for that interval. We record a *change event* when a tenant's *assigned* container size changes between successive intervals. Intuitively, a change event indicates resource demand changes equivalent to a container resize. Since we did not actually change container sizes, we can only capture cases where demand is smaller or equal to the tenant's current container size. Thus, our analysis may even be an underestimation. We use the *Inter Event Interval* (IEI), the time between two successive change events, to characterize the changes in the tenant's resource demands.

Figure 2(a) plots the cumulative percentage distribution of the IEI for all databases in the service. That is, the data point of $(60, 86)$ implies that among all container size changes detected across the service, 86% of the changes happen within 60 minutes of the previous change. From Figure 2(b) plots the percentage distribution of the number of changes per day. As is evident, more than 78% of tenants averaged at least one change event per day, more than 52% of tenants had 6 or more change events per day; and 28% of the tenants averaged more than 24 change events per day.

The key takeaway from this analysis is that an overwhelming majority of tenants in a DaaS experience significant resource demand variations (crossing container size boundaries) per day and these variations occur frequently. Therefore, an auto-scaling solution could potentially be of great value to such tenants.

2.3 Auto-scaling Knobs

For many tenants of DaaS, who are not sophisticated database administrators, it is challenging to reason about resource demands for their workload. By contrast, it is far easier for tenants to relate to application-level latency goals and monetary constraints such as a monthly budget for the database. By auto-scaling resources based on demand, we aim to bridge this gap.

In principle, an auto-scaling logic can automatically scale the container sizes purely based on demand. However, based on our analysis of various customer scenarios and talking to database administrators, we find that customers want knobs to tailor the behavior of this automated logic for their databases. A DaaS platform hosts variety of tenants, from small departmental applications with modest latency requirements to line-of-business applications requiring interactive query latency. While scaling purely based on demand might be desirable for performance-critical applications, it might not be cost-effective for budget-conscious tenants. To support a variety of tenant requirements, we expose a small set of *knobs*, which the tenants can *optionally* set, to control the costs and behavior of the auto-scaling logic.

Budget. Tenants with an operating budget for their databases can specify a budget for longer periods of time called the **budgeting period**, e.g., a month. The budget is treated as a hard constraint. When a budget is specified, the auto-scaling logic selects the smallest container sufficient to meet the resource demands and has cost less than the available budget. If the budget is not constrained, containers are selected based only on demand.

Latency goals. Applications having a latency goal for their database back-end can provide goals for the *average* or 95^{th} percentile latency. The latency goals allow the auto-scaling logic to provide resources sufficient to achieve the latency goals, thus reducing costs when resource demands require a larger container size but the latency goals can be achieved using a smaller container size. For instance, an application involving interactive user activity might specify a 95^{th} percentile latency of 100 ms. On the other hand, a small departmental web application may be content with an average latency of 1000 ms, thereby potentially achieving lower costs if the latency goals can be met with a smaller container. Latency goals of an application might not be met due to issues beyond resources, such as poorly-written application code. Therefore, latency goals are not a guarantee of performance, rather a knob to control costs.

Coarse-grained performance sensitivity. For tenants without precise latency goals, this coarse-grained knob indicates how latency-sensitive their application is. In principle, this knob can be a continuous value (e.g., between 0 and 1). However, for convenience, we expose a small set of discrete steps such as: `HIGH, MEDIUM, LOW` with the default value set to `MEDIUM`. Intuitively, for a tenant with `LOW` sensitivity, the auto-scaling logic will be less aggressive in scaling up (and more aggressive in scaling down) than for a tenant with `HIGH` sensitivity, thereby potentially reducing costs.

2.4 Solution Overview

The core of our solution is to improve the accuracy of *resource demand* estimation for a variety of SQL workloads that a DaaS serves. Figure 3 illustrates a simplified architecture of a DaaS. The service comprises a collection of servers that host the tenant databases and serve requests. Figure 3 hides other components of the service, such as the management servers, gateways and load balancers. Each database server in the service hosts a set of containers, one for each tenant database. The co-location of specific containers at a server is determined by the management fabric of the service. The DaaS collects detailed counters (called *production telemetry*) for each container.

The auto-scaling module has three major components. (a) The **Telemetry manager** (Section 3) collects telemetry for each tenant database. Its challenge is to select a subset from the hundreds of counters in the telemetry which can be used as signals to robustly-estimate demand. (b) The **Resource demand estimator** (Section 4) uses the signals from the telemetry manager to estimate the demand for each of the resources comprising the container. Signals collected from telemetry are at best weakly-predictive of resource demands. The main challenge is to accurately and robustly estimate demands for a variety of workloads. (c) The **Budget manager** (Section 5) is responsible for judiciously translating the budget for the budgeting period into a budget for each billing interval

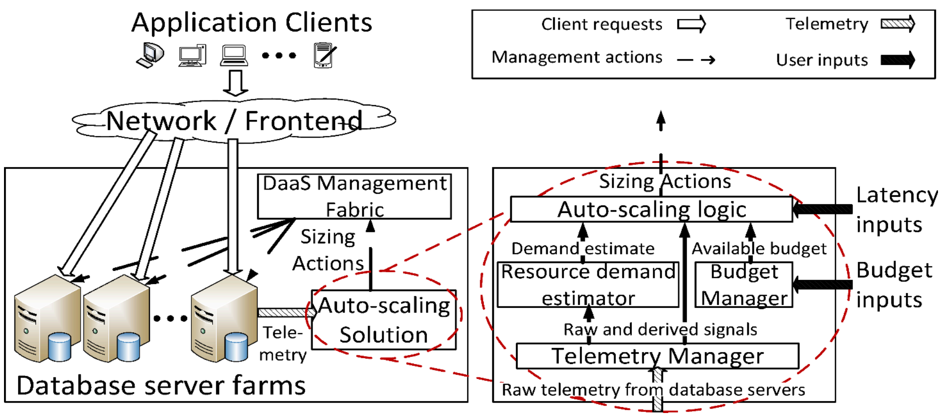


Figure 3: Simplified architecture of an auto-scaling solution for a DaaS.

for which a container is chosen. The main challenge is to design an online technique, i.e., one without knowledge of future workload patterns, which allocates enough budget for each billing interval that meets bursts in resource demands while ensuring the budget constraints for the longer budgeting period. The auto-scaling logic (Section 6) combines the three components to transform the raw telemetry and high-level tenant-specified knobs into container sizing actions supported by the DaaS, thus continuously and dynamically scaling container sizes in a closed loop as the tenant's resource demands vary over time.

3. TELEMETRY MANAGER

Mature DBMSs monitor and report hundreds of counters that comprise the production *telemetry*. The goal of the *telemetry manager* (TM) is to transform this raw telemetry into *signals* that can be used to estimate resource demands of a variety of workloads with high accuracy. First, we need to identify a small set of counters which are *predictive* of resource demand. We use domain knowledge of database engines to identify the relevant signals. As we will discuss later, these signals are at best weakly-predictive. The crux of our solution is to combine these weakly-predictive signals into robust and accurate resource demand estimation. In this section, we focus our discussion to telemetry available from Microsoft SQL Server, the engine hosting the tenant databases in Azure SQL DB. Note that many mature RDBMSs provide such telemetry. Hence, we expect many of our techniques to generalize to other systems.

Second, there is significant amounts of 'noise' in the telemetry arising from the inherent variance and spikes in workloads, transient system activities such as checkpoints interacting with workload, etc. Simple statistical measures such as *averages* can easily be 'dominated' by a few large outlier values. Therefore, as a general rule, we use statistical measures that are *robust* to outliers. We define robustness to outliers using the notion of the *breakdown point* of an estimator [18]. Intuitively, an estimator's breakdown point is the percentage of incorrect observations (e.g., arbitrarily large deviations) an estimator can handle before giving an incorrect result. *Example "breakdown point:"* Given m independent variables and their realizations $x_1, \dots x_m$, we can use $\bar{x} = (\sum_{i=1}^{m} x_i)/m$ to estimate their mean. However, this estimator has a breakdown point of 0, because we can make $\bar{x}$ arbitrarily large just by changing any *one* of the x_i. In contrast, the median of a distribution has a breakdown point of 50%, which is the highest breakdown point possible.

The remainder of this section explains the signals we obtain from the telemetry. Some signals are robust aggregates on counters obtained from the raw telemetry, while other are derived robust statistical measures over the counters.

3.1 Signals from Raw Telemetry

Latency. The database servers track latency for every request completed for a tenant. We aggregate the latencies either to the percentile or the average as specified in the latency goals.

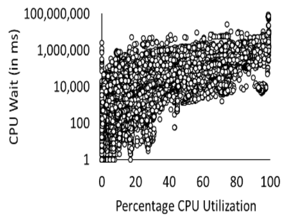
Resource Utilization. Resource utilization is a measure of how much the tenant's workload is currently using, and is an obvious signal for resource demand. The database server reports utilization for the key resources, such as CPU, memory, and disk I/O for each tenant. The TM collects these values at a fine granularity, such as once every few seconds, and computes robust statistical measures, such as the median, over different time granularities, ranging from minutes to hours, which comprise the utilization signals.

Wait Statistics. Resource utilization provides little information whether the workload needs more resources. By contrast, if a tenant's requests wait for a specific resource, it implies that the tenant has unmet demand for that resources. Mature DBMSs, such as Microsoft SQL Server, track the amount the time a tenant's request spends waiting within the database server. The TM tracks the magnitude of the wait times and the percentage waits, i.e., the time spent waiting for a resource as a percentage of the total waits for an interval of time. The TM exposes robust aggregates of these raw wait statistics as signals for demand estimation. Note that the magnitude and percentage waits are both important for demand estimation. For instance, we might observe large values of CPU waits. However, if CPU waits are insignificant compared to waits due to acquiring application-level locks, then even though there might be demand for more CPU, adding more CPU is not likely to significantly improve latencies. Similarly, CPU waits might comprise 100% of the waits. However, if the magnitude is small, then demand for CPU might not be high.

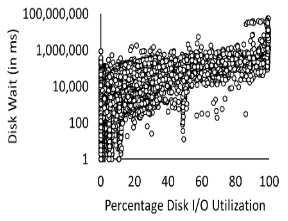
Microsoft SQL Server reports wait statistics categorized into more than 300 wait types.[6] Each wait type is associated to a (logical or physical) resource for which the request waited. Using rules, we map the wait times to the resource to determine the total time the tenant's requests waited for that resource. We classify waits into a broad set of classes for the key physical and logical resources: *CPU, memory, disk I/O, log I/O, locks,* and *system*. E.g., the *signal wait* time is the time when a thread has been signaled and ready to use to CPU to the time when it was actually allocated the CPU, and hence comprises CPU waits.

To demonstrate how wait statistics relate to and complement utilization, we analyzed the production telemetry from thousands of real tenant workloads. Figure 4 plots the CPU wait time (in ms) and Disk I/O wait time (in ms) in log scale vs. the percentage CPU utilization and percentage Disk I/O utilization respectively.

[6]http://msdn.microsoft.com/en-us/library/ms179984.aspx



(a) CPU wait ms vs. % utilization.



(b) Disk wait ms vs. % utilization.

Figure 4: Wait ms as a function of percentage utilization of CPU and Disk I/O.

Wait times are for five minute intervals; we report the median for both wait times and percentage utilization aggregated for an hour. As is evident from the figure, as the resource utilization increases, there is also an increasing trend in resource waits, though the wide "bandwidth" of the diagonal implies a weak correlation. Of particular interest is the observation that large values of resource utilization (e.g., 80) can correspond to small values of wait (e.g., $1,000$ ms), which corroborates our point that high utilization does not necessarily mean that requests are waiting due to unmet demand and hence can benefit from additional resources. Similarly, waits can be large (e.g., $1,000,000$ ms) for small utilization values (e.g., 20), which also shows that neither signal is sufficient by itself to robustly estimate demand.

3.2 Derived Signals

In addition to the (robustly-aggregated) "raw" telemetry information, we also want to analyze *trends* in latency, resource utilization etc., as well as *correlation* between signals, such as resource waits with performance. We track both correlation and trends as they serve different purposes. Trends identify changes to specific metrics (e.g., resource waits) over time, as these allow early identification of changes in the workload or its resource demands. Correlation characterizes the dependence between two signals (e.g., CPU waits and latencies). Large correlation values help us identify the main bottleneck(s) for a given workload, independently of whether there is an overall increase or decrease in these counters over time. These signals are particularly important if the tenants care about tail latencies (such as the 95^{th} percentile), since tail latencies react faster to unmet demand.

3.2.1 Robustly Identifying Trends over Time

The first class of derived signals we use is trends in a subset of monitored counters–such as latency, resource utilization, or waits–over time. For instance, if there is a trend in the recent history that the latency is degrading with time, it might be an early signal that latency goals might be violated in the immediate future. Given the immediate time scale of our resource scaling actions, we focus on detecting short-term trends with sufficient confidence. Therefore, simple, but robust, linear models are sufficient for our purposes.

The challenge in detecting trends is the noise in the underlying data on which trends are computed. The data itself might be inherently noisy and there might not be a statistically-significant trend, in which case the trend must be ignored. In addition, the trend measure should be robust to outliers.

One common approach for detecting linear trends is the use of *linear least squares* regression to identify the trend line that minimizes the squared error of the value modeled by the line and the actual values. The coefficient of determination (denoted by the R^2 error) can be used to determine how good a fit the line is for the data and reject the trend if the fit is not good. However, the least squares linear regression technique has a small breakdown point, thus making the measure unsuitable for noisy data; a single large outlier point can significantly affect the line's slope.

To address the problem, we use the Theil-Sen estimator for robust detection of the trend line [25]. Given n tuples $\langle X, Y\rangle$, the Thiel-Sen estimator computes the slope of the line passing through each pair of tuples (x_i, y_i) and (x_j, y_j) as $m_i = \frac{y_j - y_i}{x_j - x_i}$ and uses the median of m_i as the slope of the trend line. It can be shown that this estimator has a breakdown point of 29%, making it robust to significant amount of noise in the telemetry data. While there exist estimators with even higher breakdown points, the Theil-Sen estimator has the advantages of being simple, efficient to compute, and not requiring additional tuning parameters.

We use the $O(n^2)$ slopes computed by the estimator in two ways. First, the median value is the slope of any existing trend in the data. Second, we use the set of slopes to test for the existence of a significant trend in the data. That is, if there is indeed a (linear) trend in the (non-noisy) data points, then this implies that (the vast majority of) the slopes between them have the same sign. Therefore, we only 'accept' a trend if at least $\alpha\%$ of the slopes are positive or $\alpha\%$ of the slopes are negative. In our implementation, we use $\alpha = 70$, which we have found to work well in practice.

3.2.2 Robustly Detecting Correlation

If there is demand for a resource which exceeds the allocation, making the resource a bottleneck, then in the time intervals preceding that event, there is typically an increase in the utilization of that resource or the wait times associated with that resource, or both. A strong correlation between the degrading latencies and the resource utilization and/or wait counters is indicative of demand in the resource which, if met, can significantly improve latencies. Therefore, we use this correlation measure as an additional signal.

We use the Spearman rank coefficient [22], denoted by ρ, as the correlation measure. Spearman's rank correlation is a statistical measure of the correlation of ranks or orders of two ranked data sets which assesses how well the relationship between two variables can be described using a monotonic function. That is, the dependence need not be linear for Spearman's coefficient to detect it, which makes it suitable for our case since for arbitrary database applications, the correlation of utilization, waits, and latencies is often non-linear. Given two ranked data sets X and Y, ρ is computed as the Pearson's coefficient on the ranks of the x and y values. The value of ρ lies between -1 and 1; $\rho = 1$ implies perfect correlation, $\rho = -1$ implies perfect negative correlation, and $\rho = 0$ implies no correlation between the two orders. A side-effect of using the Spearman coefficient is that outliers due to data noise become much less of an issue because we first map each value to the space of ranks, which bounds the degree to which an outlier value can deviate from the average.

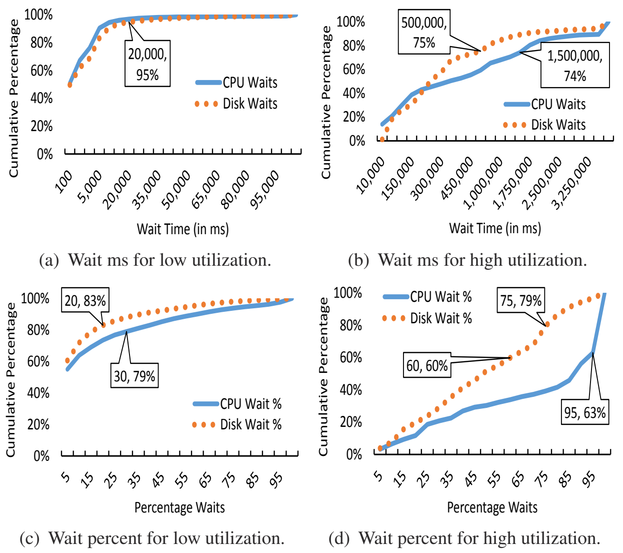
4. RESOURCE DEMAND ESTIMATOR

Each signal obtained from the telemetry manager is weakly-predictive of resource demands, and hence cannot be used in isolation to estimate demand with high accuracy. Therefore, we need a technique to combine these signals to improve accuracy of demand estimation. Our observation is that if multiple weakly-predictive signals predict high demand for a resource, it increases the likelihood of the demand actually being high.

One approach is to use statistical learning techniques to infer a model using training data obtained from production telemetry. For instance, we could use machine learning techniques to estimate resource demand from the input signals. While such an approach is elegant and was one of the first approaches we tried, we found them to be prone to over-fitting. The general problem we face for any statistical technique that depends on customer workloads is that–when collecting training data–we can only observe a very small fraction of space of the possible customer workloads. When using a sufficiently-powerful statistical technique to combine different signals from production telemetry, we found the resulting model to have high prediction accuracy on the workload it had been trained on. However, the accuracy would degrade very significantly for other, unseen workloads. This is a major challenge for a relational DaaS platform which caters to a variety of applications and workloads. The tenant is free to execute any arbitrary SQL code and user defined functions which can be very different from what the model was trained on. Similar issues with statistical techniques were also reported in Li et al. [12]. Note that we do not claim that over-fitting is an inevitable side-effect of using fully-statistical learning techniques; they may be overcome with careful selection of features, the learners, and training data, something we would like to continue exploring in the future.

Our approach is to combine signals by leveraging domain knowledge of the internals of the database engine and how different resources interact. We propose a decision logic comprising a manually-constructed hierarchy of rules that use multiple signals to determine the resource demands of the tenant's workload. The core of the rule-based logic is a set of thresholds for each signal to determine the tenant's *state* in terms of each signal in isolation. Each rule combines the states obtained from each signal to determine the overall resource demand. By traversing this hierarchy of rules, the logic decides to add more resources (i.e., scale-up) if there is high demand or take away unused resources (i.e., scale-down) if the demand is low. Recall that given the discrete container sizes and that at any instant of time, the tenant is associated with a container size, our problem is to estimate if there is demand for a larger container or the demand can be met by a smaller container. When estimating demand, we determine in each resource dimension, how many steps in container sizes do we need to increase (or decrease). We use production telemetry across thousands of tenants to guide us through this process. By assigning container sizes to tenant's resource utilization values (similar to that in Section 2.2), we observed that of the total number of container size change events due to resource demands changing, 90% result in the container size changing by 1 step, and step sizes 1 and 2 together comprise 98% of the changes. Therefore, we constrain our problem to estimating demand to change container size by 0 (i.e., no change), 1, or 2 steps.

There are several pragmatic benefits of our approach. First, we found it to be robust when testing across a wide range of very different workloads. Second, once thresholds are applied to the signals, it transforms the signals from a continuous value domain to a categorical value domain where each category has easy-to-understand semantics. This makes the rules easier to construct, debug, maintain, extend, and explain. For instance, using categories with well-defined semantics allows the auto-scaling logic to provide an *"explanation"* of its actions. These explanations provide the (often unsophisticated) end-users with a simple reasoning for scaling actions. The container sizing decisions result from analyzing tens of signals. However, the model traverses a hierarchy of rules with well-understood semantics for each path. An explanation is a concise way of explaining the path the model traversed when recommending a container size. For instance, if the model identifies a CPU bottleneck which in turn scales up the container size, then an explanation of the form *"Scale-up due to a CPU bottleneck"* is generated. If the model recommended a scale-up but the budget constraints disallow it, then it can generate an explanation of the form *"Scale-up constrained by budget."* For an expert administrator, the model also exposes the raw telemetry as well as the signals used to facilitate detailed diagnostics.



(a) Wait ms for low utilization. (b) Wait ms for high utilization.

(c) Wait percent for low utilization. (d) Wait percent for high utilization.

Figure 6: Distribution of wait ms and wait time for a resource as a percentage of total waits for CPU and Disk I/O.

We now explain the steps in resource demand estimation: determining the thresholds for each signal and how we leverage production telemetry from a DaaS to determine these thresholds, and some example rules to detect high and low demand.

4.1 Determining Thresholds

Setting thresholds for latencies and resource utilization is straightforward; Figure 5 illustrates this. If the tenant specifies a latency goal, it becomes the threshold to determine if the latency is `GOOD` (i.e., goals are met) vs. it is `BAD`, i.e., goals are not being met. Similarly, for the underlying database engine the DaaS platform utilizes, there already exist well-known thresholds and rules that system administrators use to categorize the resource utilization as `LOW`, `MEDIUM`, and `HIGH` (e.g., see Figure 5).

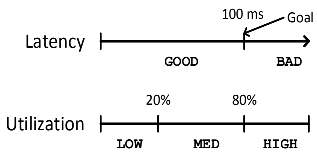


Figure 5: Categorizing signals using thresholds.

Waits associated with a resource can also be categorized as `HIGH`, `MEDIUM` or `LOW`. However, deriving thresholds to categorize waits is not as straightforward as latency and resource utilization. As can be seen from Figure 4, resource waits can be as large as $1,000$ seconds even with low utilization levels of 20% or 30% and waits can be as low as 1 second even for high utilization levels of 70% or 80%. Similar reasoning also applies for the percentage waits. It is imperative that we use a systematic approach for setting thresholds for the wait statistics in order that the categories are meaningful. We use production telemetry collected from thousands of real tenant's databases across the service to determine these thresholds. The rationale behind our approach is that if resource demands are high, wait for that resource will also be high. Since this correlation is weak, there can be occasional noise. However, if we analyze data from thousands of tenants, there should be a clear separation between waits for low and high demand; our analysis of production telemetry supports this hypothesis.

The first challenge is in determining the thresholds to categorize the resource wait times into `HIGH`, `MEDIUM`, and `LOW`. Figures 6(a) and 6(b) plot the cumulative distribution of the wait times (in ms) for CPU and Disk I/O for different levels of utilization for the corresponding resource. Wait times reported are for five minute intervals and we report the median over an hour. We separate the waits based on the utilization levels (low and high) of the corresponding resource, which we use as a proxy for high/low demand. Resource utilization is high if the average hourly utilization of that resource is more than 70% and low if the average hourly utilization is less than 30%. When the resource utilization is low, even the 90^{th} percentile of both CPU and Disk I/O waits is about 20 seconds (see Figure 6(a)). On the other hand, when the resource utilization is high, the 75^{th} percentile of Disk I/O waits is 500 seconds and that for CPU waits is 1500 seconds. Therefore, there exists a clear separation between the wait distributions for high and low utilization. We use percentile values from these distributions to categorize waits as `HIGH`, for instance if CPU waits exceed 1500 seconds, or `LOW` if it is less than 20 seconds. The percentiles shown in the figure are for illustration, the actual percentile is different for each container size, resources type, and cluster configuration.

The second challenge is to identify the thresholds for percentage waits to be `SIGNIFICANT` or `NOT SIGNIFICANT`. Again, we use production telemetry to set thresholds for percentage waits. Figures 6(c) and 6(d) plot the distribution of the percentage waits corresponding to CPU and Disk I/O. Similarly to the previous case, we plot separate distributions for low and high utilization levels for that resource. As is evident from Figure 6(c), the 80^{th} percentile of percentage waits for CPU and Disk I/O is in the range $20\% - 30\%$ while the corresponding number for high utilization is in the range $70\% - 90\%$ which demonstrates this separation in values.

As the software evolves, new hardware SKUs are deployed in the data centers, and new container sizes are supported in the service, these thresholds need to be re-tuned. Updating these thresholds incrementally is automated through reports and alerts expressed over the aggregate telemetry collected from the service.

4.2 Detecting High Demand

We now explain how we use our knowledge of the database engine internals to craft a set of rules using the signals to estimate whether demand is high enough to require a scale-up. The first step is to identify the scenarios that correspond to high demand. A few *illustrative scenarios* are: (a) If utilization is `HIGH` and wait times are `HIGH` with `SIGNIFICANT` percentage waits. (b) If utilization is `HIGH`, wait times are `HIGH`, percentage waits are `NOT SIGNIFICANT`, and there is a `SIGNIFICANT` increasing trend over time in utilization and/or wait. (c) If utilization is `HIGH`, wait times are `MEDIUM`, percentage waits are `SIGNIFICANT`, and there is a `SIGNIFICANT` increasing trend over time in utilization and/or waits. Note that all of the scenarios combine two or more signals. Moreover, if one of the signals is weak (e.g., wait time in not `HIGH`), we consider additional signals (e.g., trends).

Note that we stated the scenarios in terms of the signals and their categories. In addition to being easy to explain, these scenarios can be directly encoded as predicate rules in the model which if true for a resource implies high demand for that resource.

Further note that memory and disk I/O interact. That is, if memory is a bottleneck, it will result in higher I/O utilization and waits. Since memory waits and tracked independent of I/O waits, if both resources are identified as a bottleneck, the model will recommend scaling-up both resources.

4.3 Detecting Low demand

Estimating whether demand is low is similar to high demand estimation, except that the tests are for the other end of the spectrum of categories for the signals. For instance, the rules test for `LOW` utilization or `LOW` waits, and non-positive trends in resource waits or utilization; we omit the rules for brevity. However, an interesting case is detecting low memory demand, which we discuss in the remainder of this subsection.

Memory utilization of a database server is rarely `LOW`. This is because the largest consumers of memory in a database server are the different caches, such as the buffer pool and the query plan cache, which do not voluntarily release memory. When all pages accessed are in memory, the waits associated with memory are also `LOW`. Therefore, estimating low memory demand using memory utilization and waits alone is hard since these signals do not accurately differentiate low demand from the case where there is demand but all requests are being met using memory already allocated to the tenant. For instance, if the tenant's working set fits in memory, it will result in low memory waits. However, when memory is reduced to a point where the working set does not fit in memory, there will be a significant increase in memory demand and subsequent impact on latency. Therefore, in addition to low memory waits, we must also account for the potential increase in number of disk I/Os and memory wait times as a result of reducing memory. Memory demand is low only if the expected increase in disk I/Os and wait times is below a threshold. Estimating this expected increase in I/Os when reducing memory for arbitrary database workloads is incredibly challenging.

We use a technique inspired by ballooning [3, 27] where we slowly reduce the memory allocated to a tenant to observe its impact on disk I/O. If the memory can be reduced all the way to the next smaller container size without causing a significant increase in disk I/O demand, we determine memory demand as low. If ballooning results in increase in disk I/O demand, we revert the tenant to the current memory allocation. Our challenge is to determine *when* to trigger ballooning so that the impact of latencies can be minimized. We trigger ballooning only when the demand for all other resources is `LOW`. While being conservative, in the absence of an accurate and versatile model to predict the impact of reducing memory on latencies, this strategy minimizes the risk of adversely affecting the tenant's query latencies.

5. BUDGET MANAGER

A tenant can specify a budget ($\mathbb{B}$) for a *budgeting period* comprising n billing intervals. The budget manager needs to determine the *available budget* (B_i) for each billing interval, which is considerably smaller than the budgeting period, such that $\sum_{i=1}^{n} B_i \leq \mathbb{B}$. Let C_{min} and C_{max} be the respective costs per billing interval for the cheapest and the most expensive containers. The budget manager must ensure $B_i \geq C_{min}$ in order still be able to allocate the cheapest container within the budget. The main challenge is in managing the surplus budget ($\mathbb{B} - n \times C_{min}$). The simplest approach is to spread the surplus equally among each interval. However, tenants' resource demands and workloads arrive in bursts, thus making the even-spread approach unsuitable. Moreover, since the budget allocation for an interval is online without any knowledge of the future workload arrival patterns, the allocation algorithm must balance meeting current demands while retaining sufficient budget for unanticipated bursts in future intervals.

We draw analogy of our budget management problem to the traffic shaping problem in computer networks [24]. A network router shapes a flow, allowing periodic bandwidth bursts, while providing a minimum steady bandwidth, and ensuring that the flow conforms to a total bandwidth allocation. We adapt the *token bucket algorithm* [24] for our problem of budget allocation. Figure 7 illustrates this algorithm which uses a fixed capacity bucket, of *depth* D, to hold tokens where D is the maximum burst size. The bucket is initialized with T_I tokens and periodically refreshed at a fixed rate, called the *fill rate* T_R, which is the guaranteed average bandwidth.

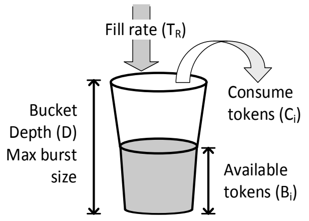


Figure 7: Token bucket.

We now describe how the budget manager configures the token bucket by setting T_R, T_I, and D to meet the requirements. At any instant, the number of tokens in the bucket is the available budget B_i. At the end of the i^{th} billing interval, T_R tokens are added and C_i tokens are subtracted, where C_i is the cost for the i^{th} interval. Setting $T_R = C_{min}$ ensures that $B_i \geq C_{min}$. Setting $D = \mathbb{B} - (n-1) \times C_{min}$ guarantees $\sum_{i=1}^{n} C_i \leq \mathbb{B}$. The value we set for T_I determines how aggressive we are in consuming the budget during periods of high demand. An aggressive bursting strategy sets $T_I = D$, i.e., start the budgeting period with a full bucket. If there is a sustained high demand such that the largest container is allocated for m intervals, the bucket will be empty when $m = \frac{\mathbb{B}-(n-m)\times C_{min}}{C_{max}}$. Starting from the $(m+1)^{th}$ interval to the n^{th} interval, the available budget will be $B_i = C_{min}$ and the tenant can only use the cheapest container which might not be enough to meet the demands. An alternative is to set $T_I = K \times C_{max}$, where $K < m$ and set $T_R = \frac{\mathbb{B}-T_I}{n-1}$. This conservative setting ensures that the maximum usage burst is limited to at most K intervals of using C_{max} plus any surplus unused tokens unused from the past, i.e., this setting saves more for intervals later in the budgeting period at the expense of limiting costs early on. A service administrator can analyze the production telemetry to set a suitable value of K.

6. AUTO-SCALING LOGIC

The auto-scaling logic determines the container size for the next billing interval by monitoring latencies, resource demands, and the available budget. At the end of a billing interval, the auto-scaling logic determines if the tenant has high demand for a resource. If the latency is `BAD`, or there is a `SIGNIFICANT` increasing trend of latency with time, then the logic will recommend scaling-up if enough budget is available. If latency is `GOOD` and not degrading, and resource demands are `LOW`, then the logic recommends scale-down. Otherwise it takes no action.

The resource demand estimator determines if more (or less) of each resource resource is desired. The resource demand of each resource comprises the desired container size. The auto-scaling logic uses the available budget (B_i) and the desired container size to find the cheapest container, among the set of DaaS's containers, with resources greater or equal to the desired container on all resource dimensions and price $C_i \leq B_i$. If desired container is constrained by the available budget, then the most expensive container with price less than B_i is selected. This process is an iterative search over the set of containers supported by the DaaS. Since the resource demand estimation is for individual resources. Therefore, if the workload only has demand for one type of resource, such as CPU, then the estimation logic will only recommend increasing the CPU allocation in the desired container. If the DaaS supports scaling containers in each resource dimension, for instance Figure 1, the auto-scaling logic can leverage that.

If the container size recommended is different from the current container size, the model issues a container resize command to the management fabric of the DaaS which then executes the resize operation. This container resize operation is an online operation.

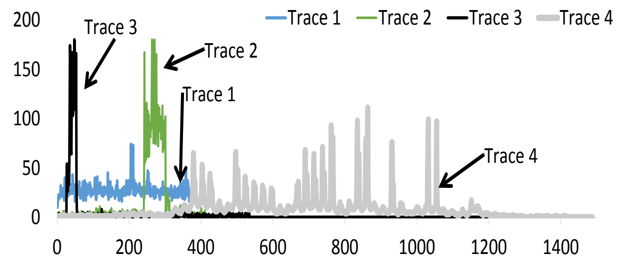


Figure 8: Traces derived from real-life workloads used for the experiments. Horizontal axis is time in minutes, vertical axis is number of concurrent requests per second.

7. EXPERIMENTAL EVALUATION

We prototyped our auto-scaling solution in Azure SQL Database which supports a set of fixed sized containers. In this section, we experimentally evaluate the effectiveness of our approach (henceforth referred to as **Auto** for brevity) using a variety of workloads and resource demand patterns derived from production traces. We compare our approach with a number of alternatives solution using **latency** (the 95th percentile latency of the workload), and **cost** incurred on behalf of the tenant. These experiments demonstrate that Auto's improved demand estimation accuracy results in significantly lower costs (in the range of $1.5\times$ to $8\times$) while achieving the desired latency goals for a wide variety of workloads and resource demand patterns, which also demonstrates its robustness.

7.1 Methodology

Our technique estimates the resource demand characteristics exhibited by any SQL workload and does not make any assumptions about the knowledge of the workload. We, therefore, expect our solution to to be effective for both transactional as well as data warehousing workloads. For brevity, we focus our experiments here on transactional workloads with time-varying resource demands. We use demand patterns derived from production workloads to control the load for some well-known benchmarks to create time-varying resource demands. Figure 8 plots four *"traces"* derived from real customer workloads in a production Daas. In each figure, the horizontal axis is time and the vertical axis is number of concurrent requests in the real workload.

Each trace is chosen to target a specific demand scenario. Trace 1 corresponds to a workload with steady demand. This trace is suitable for a static container size and the goal is to validate that an auto-scaling logic is at least competitive to a technique that knows the demand upfront and sets a static container size suitable for the demand. Traces 2 and 3 correspond to workloads which are mostly idle with one burst of resource demand; in trace 2 the burst lasts longer compared to trace 3. These demand patterns are suitable for an auto-scaling solution that reacts to the burst, scales up the container size during the period of high demand, and subsequently scales-down to save cost. Trace 4 exhibits lots of bursts in demand for short intervals. This workload is intended to stress-test the online auto-scaling solutions.

We use standard benchmark workloads to generate the actual user requests. The workload generator executes in steps in sync with the trace. At every step, the workload generator reads the number of requests from the trace to set the target number of requests/sec for the workload until the next step. At every step, the workload generator executes transactions from the benchmark workload and maintains the offered load as close as possible to the specified target. The workload generator executes the entire trace which comprises one run for that experiment. We use three types of workloads in our experiments: TPC-C, Dell DVD Store

(DS2),[7] and a synthetic micro-benchmark (CPUIO) that generates queries that are CPU-, disk I/O- and/or log I/O-intensive. Our goal of selecting these workloads is to generate a variety of transactions/query mixes ranging from short read/write transactions to lightweight analytical queries that scan parts of the data and computes aggregates. The CPUIO micro-benchmark allows us to execute queries that create demand for each of CPU, memory, and I/O while allowing us to alter the mix of the queries. The workload's working set is controlled by creating a hotspot in data accesses. This variety helps us evaluate the robustness of the solutions.

A tenant is an instance of one of these workloads and connects to its own database enclosed in a container. The auto-scaling module monitors the tenant's workload demands and sets the appropriate container size for the billing interval. We use a set of eleven container sizes modeled similar to ones supported by today's commercial offerings such as Microsoft Azure SQL DB, Amazon RDS, and Google Cloud SQL. The container sizes cover a large gamut of resource allocations all the way from half-a-core of CPU allocation for the smallest container to tens of CPU cores for the largest container. Other resources, such as memory and I/O, also have a similar spectrum; the cost of a container ranges from 7 units to 270 units for each billing interval.

For the purposes of experimentation, we compress the time scales for each workload trace as well as the billing interval. This compression makes the problem more challenging for an auto-scaling since changes in demands are more drastic. It also allows us to observe many instances of the container size changes without requiring to run the experiment for days. We use minutes as unit of time for the horizontal axis of each trace in Figure 8. This implies that an experiment on a trace takes from a few hours to up to a day depending on the length of the trace. We set the billing interval to one minute. Note that having a billing interval of a minute does not imply the auto-scaling module will change container sizes every minute—the auto-scaling module changes container sizes only when the demand changes, which we validate using the number of container size changes during an experiment.

7.2 Alternative Solutions

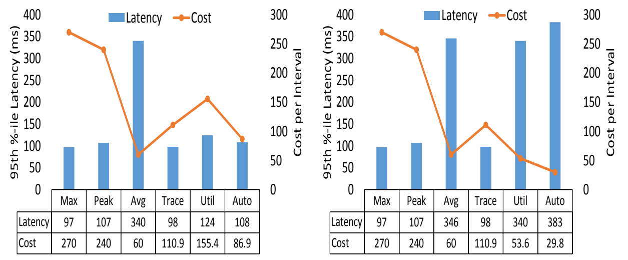
We compare Auto against a set of practical solutions which can be implemented in production today without requiring a deep understanding and knowledge of the application's workload. We compare Auto with two offline and one online solution. We also compare against a setting (called **Max**) where we statically set the tenant's container to the largest container supported by the DaaS. This results in the highest cost, but also provides a *gold standard* with the *best* performance for the workload.

7.2.1 Offline solutions

We use two offline solutions which have the luxury to observe the resource demands of the workload trace before making a choice of the container size. Since these techniques cannot dynamically adapt container sizes based on latency goals, we do not set any latency goal for these techniques. Once an offline solution selects a container size, we replay the exact same workload to measure the impact of container choice on latency.

Static. This solution simulates an approach a typical administrator who has knowledge about the historical resource demands for its application which s/he uses to statically set the container size. We execute the workload with **Max** to analyze the resource utilization and then set the container size to be the smallest container that can meet the historical utilization. We experiment with two settings for the container size: one using the 95^{th} percentile resource utilization for the entire workload to configure the container for the **Peak**, and one using the *average* utilization to configure for the **Avg**.

[7] http://linux.dell.com/dvdstore/



	Max	Peak	Avg	Trace	Util	Auto
Latency	97	107	340	98	124	108
Cost	270	240	60	110.9	155.4	86.9

(a) Goal: $1.25\times$ **Max** = 120 ms

	Max	Peak	Avg	Trace	Util	Auto
Latency	97	107	346	98	340	383
Cost	270	240	60	110.9	53.6	29.8

(b) Goal: $5\times$ **Max** = 485 ms

Figure 9: The impact of latency targets to trade costs with latency for a bursty resource demand (Trace 2) executing the CPUIO micro benchmark workload.

Trace. This solution simulates an offline approach which exactly *"knows"* the workload's resource demands and can generate a sequence of container size changes that *"hugs"* the resource utilization and demand curve. We generate this trace of hugging containers by executing the workload with **Max** to obtain the actual resource utilization of the workload. For a given billing interval, we then select the smallest container that meets the resource requirements for that interval. The output trace is of the form $\langle i, \text{Cont}_i \rangle$ where i is the billing interval and Cont_i is the container size for interval i. The exact workload is then repeated; during replay, at every interval i, the container size is set to Cont_i.

7.2.2 Online solutions

An online solution, such as Auto, estimates the resource demands and monitors performance to determine the container size. For these solutions, we specify a latency goal for the entire workload. We derive the latency goal from the 95^{th} percentile latency measured by running the workload with **Max**, which corresponds to best latency. In one setting, we use $1.25\times$ latency with **Max** as the goal, and in another we use $5\times$ the latency with **Max** as the goal. Our evaluation primarily focuses on how the online solutions change container sizes based on resource demands and latency. We set the budget to the default value since it is intended as a tenant-specified safeguard and is not of interest for these experiments; experiments with budget constraint is omitted for brevity.

We compare against an online auto-scaling solution (**Util**) which uses the latency and the utilization of every resource to decide the container size. This solution emulates the auto-scaling offerings for VMs that today's cloud providers support, translated to our setting of determining container sizes instead of adding or removing VMs. This solution tracks latencies, and if latency is `BAD` and resource utilization is `GOOD` or `HIGH` then scales-up the container size. On the contrary, if performance is `GOOD` and resource utilization is `LOW`, it scales-down the container size.

7.3 End-to-End Results

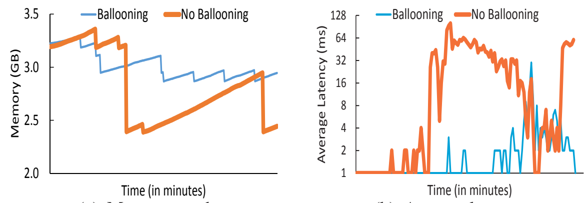
We now report end-to-end results using a variety of resource demand traces and workloads. For the online techniques, latency goals are set for the 95^{th} percentile for the entire workload; the offline techniques to not consider latency goals. Recall that use **Auto** to refer to our solution.

Figure 9 plots the cost and latencies for running the CPUIO workload using the demand pattern from Trace 2. The bars report the 95^{th} percentile latency for the workload (in ms) and is plotted along the primary (left) vertical axis. The line reports the average cost per billing interval for each technique and is plotted along the secondary (right) vertical axis. The different sub figures correspond to the different setting for the latency goals. Figure 9(a) reports the experiment where the latency goal was set to $1.25\times$ the latency with Max (i.e., 120 ms) and Figure 9(b) reports the experiment where the latency goal was set to $5\times$ the latency with Max (i.e., 485 ms). Recall that Trace 2 has one big burst of resource demand which is preceded and followed by periods of low activity. The goal of this experiment is to demonstrate that a good auto-scaling solution can result in *significant* cost savings while achieving good latencies. As is evident from Figure 9(a), when the tenant desires latencies close to that with Max, Auto helps the tenant achieve the latency goal with $2.75\times$ lower cost when compared to allocation for the Peak (i.e., 95^{th} percentile of resource utilization). Note that provisioning for Avg results in lower costs, but the latency is almost $3\times$ worse compared to the goal. The other online solution Util results in about $1.8\times$ higher cost compared to Auto. This demonstrates that reasoning about resource demands using multiple signals, such as utilization, resource waits, trends, and correlation can significantly lower costs of Auto while achieving good latencies within the tenant-specified goals. While not directly comparable, putting the results in perspective with the offline technique Trace, Auto results in a lower costs, though Trace achieves a slightly better latency. However, if the objective is achieve the least cost while meeting the goal, Auto's lower cost trumps Trace.

When the latency goal is set to $5\times$ that with Max (Figure 9(b)), Auto results in $2\times$ lower costs compared to Avg, about $8\times$ lower costs compared to Peak, and about $1.8\times$ lower cost compared to Util, all while meeting the tenant-specified latency goals. This experiment validates our claim that when the tenant's latency goals are less stringent, Auto reduces monetary costs even further. In terms of the number of container size changes, both Auto and Util made container size changes for about 11% of the total number of billing intervals during the experiment. In contrast, Trace, which hugs the demand curve, results in container size changes in about 15% of the billing intervals.

Figure 10 plots the latency and cost for the TPC-C workload using the demand pattern from Trace 4. In this experiment, the latency goal is set to $1.25\times$ that of Max (i.e., 340 ms). Similar to the previous results, Auto results in significantly lower costs compared to all other alternatives while meeting the latency goal. Among the the techniques meeting the latency goal, Peak costs $2\times$ more, Trace costs about $2.4\times$, and Util costs about $3.4\times$ that of Auto.

To understand why Util costs $3.4\times$ that of Auto, we report the container sizes, resource utilization, and the *performance factor* (which is the observed latency as a percentage of the goal, negative values imply the goal is not met) for both Util (Figure 13(a)) and Auto (Figure 13(b)). For ease of exposition, we only plot the CPU settings for the container and the CPU utilization (plotted along the primary vertical axis) since it is the dominant resource for this workload; the performance factor is plotted along the secondary vertical axis. Both CPU utilization and the container's Max CPU are expressed as percentage of the total server capacity on which the container is hosted. For both techniques, the performance factor is close to zero, which implies that performance is close to its goal. Both techniques are designed to keep a buffer for performance so that while they react to unanticipated resource demands, the tenant's performance does not significantly degrade beyond the goal. Therefore, both techniques will periodically scale up reacting to high resource demands and compensating for performance degradations for the period it takes to react to changes in demand. However, note that when Util decides to scale up, it ends up scaling much higher to compensate for the latency degradation—we see the container sizes as high as 70% of the server's CPU capacity. On the other hand, Auto's container size selection is in the range of 10% to



(a) Memory used (b) Average latency.

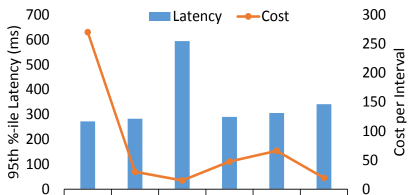
Figure 14: The impact of ballooning on end-to-end latency for CPUIO micro benchmark workload with a steady demand.

20% of the server's resources. Observe that even though the container sizes are large for Util, the CPU utilization continues to peak at about 10% for both techniques. This is because for this workload, the majority of time is spent on acquiring application-level locks and hence the workload's latencies cannot significantly benefit from the additional resources. This is evident from Figure 13(c) which reports the percentage waits for each wait category. Observe that lock waits (which are in a dark fill color) are more than 90% of the waits and dominate any other resource wait category. Auto uses the wait statistics and the correlation between resource waits and latencies to identify this bottleneck beyond resources and does not unnecessarily add resources. Based on this analysis, Auto selects smaller container sizes for most intervals which is enough to meet the resource demands for the workload while ensuring the latency remains close to the specified goal. This experiment demonstrates how domain knowledge of the internals of the database engines, using multiple signals, and reasoning about resource demand can significantly reduce costs while meeting the latency goals.

Figure 11 reports the latency and costs for the CPUIO workload with resource demand pattern driven by Trace 3 which is also a bursty workload. The latency goal is set to $5\times$ Max (i.e., 500 ms). The benefits of Auto is also evident in this scenario where Peak incurs $4.5\times$ higher cost, Avg incurs $1.5\times$ higher cost, and Util results in $2.5\times$ higher cost. Finally, Figure 12 reports the cost and latency for DS2 workload with an almost steady demand pattern driven by Trace 1. Even for this workload, which is perfect for a static container size, Peak incurs $1.5\times$ higher cost, Avg incurs $1.2\times$ higher cost, and Util incurs $1.5\times$ higher cost compared to Auto. This demonstrates that not only is Auto beneficial for bursty workloads, it also results in significant reduction in costs while meeting the latency goals even for workloads with low variance in demand. These experiments demonstrates Auto excellent performance and robustness with a variety of workloads and resource demand patterns.

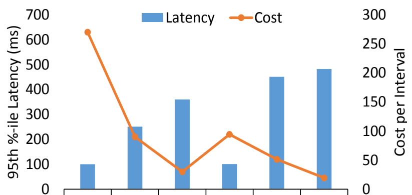
7.4 Ballooning and low memory demand

This experiment demonstrates the impact of using ballooning to avoid significant impact on the tenant's end-to-end latency due to incorrect estimation of low memory demand. We use the CPUIO workload with Trace 1 to generate a steady demand for all resources. The working set is close to 3GB and is controlled using a *hotspot* access distribution with more than 95% operations accessing data in the working set. Figure 14(a) plots the memory used by the tenant (in GB) as time progresses. The two lines correspond to Auto with and without ballooning. Once the workload reaches a steady state, the working set is cached in memory (with used memory $> 3GB$). Since the working set fits in memory, this corresponds to the hard case of detecting low memory demand. In the absence of ballooning, Auto changes the memory allocation to the next smaller container size (2.5GB) which causes a sharp drop in used memory (see Figure 14(a), thick orange line). This change causes the query latencies to increase two orders of magnitude; Figure 14(b) plots the average end-to-end query latency (in ms, log



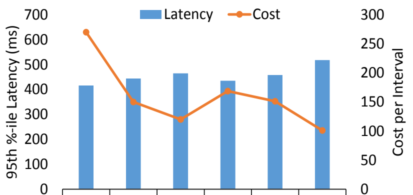
	Max	Peak	Avg	Trace	Util	Auto
Latency	272	283	594	290	306	341
Cost	270	30	15	47.4	66.1	19.5

Figure 10: TPC-C, Trace 4, Latency goal: $1.25\times$ **Max =** 340 **ms.**



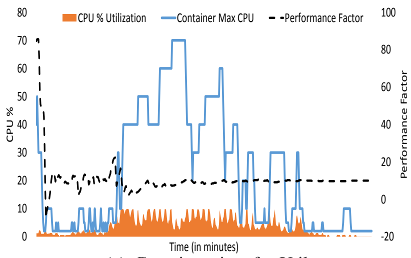
	Max	Peak	Avg	Trace	Util	Auto
Latency	100	251	360	101	451	482
Cost	270	90	30	94.3	51.4	19.5

Figure 11: CPUI0, Trace 3, Latency goal: $5\times$ **Max =** 500 **ms.**

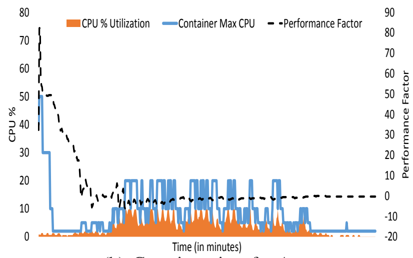


	Max	Peak	Avg	Trace	Util	Auto
Latency	416	444	465	435	458	518
Cost	270	150	120	168.8	151.2	101

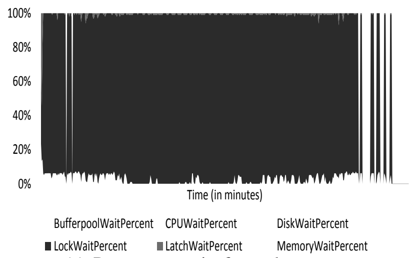
Figure 12: DS2, Trace 1, Latency goal: $1.25\times$ **Max =** 520 **ms.**



(a) Container sizes for Util



(b) Container sizes for Auto



(c) Percentage waits for each category

Figure 13: Drill down explaining the behavior of Util and Auto as time progressed through the experiment.

scale) as time progresses. Auto notices this increase in latency due to unmet disk I/O demand and reverts to the larger container. However, it takes a long time for the working set to be entirely cached in memory, which prolongs the impact on latency. By contrast, with ballooning, Auto starts the process of gradually reducing memory. However, it aborts before reaching the next smaller container, since it notices increased I/Os around the point where memory is reduced to $3GB$, which corresponds to the working set. Therefore, Auto equipped with ballooning results in minimal latency impact in the event of inaccurate low memory demand estimation.

8. RELATED WORK

Resource estimation, provisioning, and auto-scaling resources has been an active area of research in the database and the systems research communities. Existing approaches can be classified into black-box application-agnostic solutions, typically used for auto-scaling VMs that host a variety of applications, white-box and black-box modeling for database workloads typically for enterprise consolidation scenarios, and models for estimating resources for individual long-running data analysis queries.

There is a large body of work on application-agnostic techniques to auto-scale resources for any application running inside a VM such that a certain set of application SLOs can be achieved. Examples include CloudScale [19], AGILE [16], PRESS [9], AutoControl [17], AutoScale [8], CloudStone [13], SCADS [26], and Huber et al. [11]. Since these techniques are application-agnostic, they must rely on OS/hypervisor level counters to detect resource utilization and pressure. In contrast, our focus is on auto-scaling resources in a database server. As we show in this paper, by exploiting database-specific signals and building white-box models based on domain knowledge of database engine, we are able to significantly improve the accuracy in demand estimation, thereby lowering costs while achieving query latencies similar to approaches using generic resource utilization counters.

Resource modeling for database workloads in an enterprise workload consolidation scenario has also been studied. These solutions rely on a "representative workload" for each application to be provided upfront, which is then used to construct models that estimate the resources for that workload. Soror et al. [21] use the query optimizer's 'what-if' mode to estimate the impact of a particular resource on the performance. Kairos [3] replays the workload in a separate sandbox environment to model the resource utilization and demands. DBSeer [14] builds models to predict the resource utilization for a given set of transactions in an OLTP workload which can then be used to estimate resource requirements as the throughput of the workload changes (e.g., change in disk I/O if the throughput doubles). In contrast, since a DaaS is a platform to which tenants can and do submit arbitrary and ad-hoc queries, we are unable to make a closed-world assumption about knowledge of a "representative workload" to train our model. Thus, our resource demand estimation logic does not make assumptions that are specific to a workload, such as transaction types, or the fraction of each transaction type, and relies instead on exploiting correlation between lower-level counters that are generic to any database workloads. We do not require running the workloads in a sandbox environment, which is often infeasible in a DaaS setting. Due to the complexity of the problem and the inability to make a closed world assumption, instead of accurately predicting the resource requirements of a specific workload mix, we focus on a different problem of estimating if the current mix of workloads has demand a larger (or a smaller) container size in each resource dimension. Different from resource modeling, Pythia [6] targets the problem of modeling which combinations of tenants can be co-located at a server in a setting where tenant's workloads are not isolated using containers. The goal is to identify combinations of tenants which can be co-located without affecting each other's performance.

Accurately predicting resource usage for individual long-running queries is another area of active research [1, 2, 5, 7, 12]. These approaches typically rely on training statistical models on actual observations of resource consumptions for training queries executed in isolation. These models find applications in resource management and query scheduling in data warehouses with a fixed set of resources executing a handful of concurrent queries. By contrast, a typical tenant of a DaaS often executes hundreds of concurrent short-running queries typically finishing within milliseconds to seconds. The per-query models targeted a different setting and it is extremely challenging to generalize these models to estimate the resource requirements for the sum total of hundreds of concurrently executing queries. Instead of accurately estimating the resource requirements of tens or hundreds of concurrent queries, we focus on coarser-granularity estimation of whether the current workload mix has demand for the next larger (or smaller) container.

9. CONCLUDING REMARKS

Elasticity and pay-per-use are two key attractions of cloud platforms. Tenants can leverage this elasticity to reduce costs during periods of low demand and improve latencies during periods of high demand. We explored this problem of how a DaaS provider can efficiently and robustly support auto-scaling container sizes on behalf of its tenants. We demonstrated that resource utilization alone is not a good estimator for demand for a variety of database workloads. By using domain knowledge of database engines, we proposed a more accurate demand estimator that uses a statistically-robust set of signals, a decision logic to combine multiple signals, and a way to leverage the service-wide telemetry across thousands of tenants of a DaaS. Using a more accurate demand estimator coupled with optional tenant-specified inputs of monetary budget constraints and latency goals, we described an end-to-end auto-scaling solution for a DaaS. We prototyped our approach in Microsoft Azure SQL Database and demonstrated the cost effectiveness of our solution and robustness using a wide variety of benchmark workloads and resource demand traces derived from production. Our solution raises the abstraction for tenants of a DaaS by allowing them to reason about monetary budget and query latency rather than resource provisioning.

Acknowledgements

The authors would like to thank the anonymous reviewers, Pamela Bhattacharya, and Ishai Menache for useful feedback to improve this paper, Manoj Syamala and George Reynya for useful feedback to better understand the database engine internals, and Kukjin Lee for the help in analyzing production telemetry. Several members of the Microsoft Azure SQL Database group, including Peter Carlin, Miroslav Grbic, Veljko Vasic, and Petar Lotrean, have provided feedback that influenced our work.

10. REFERENCES

[1] M. Ahmad, S. Duan, A. Aboulnaga, and S. Babu. Predicting completion times of batch query workloads using interaction-aware models and simulation. In *EDBT*, pages 449–460, 2011.

[2] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based query performance modeling and prediction. In *ICDE*, pages 390–401, 2012.

[3] C. Curino, E. P. C. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *SIGMOD*, pages 313–324, 2011.

[4] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Trans. Database Syst.*, 38(1):5, 2013.

[5] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal. Performance prediction for concurrent database workloads. In *SIGMOD*, pages 337–348, 2011.

[6] A. Elmore, S. Das, A. Pucher, D. Agrawal, A. E. Abbadi, and X. Yan. Characterizing Tenant Behavior for Placement and Crisis Mitigation in Multitenant DBMSs. In *SIGMOD*, pages 517–528, 2013.

[7] A. Ganapathi, H. Kuno, U. Dayal, J. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *ICDE*, pages 592–603, 2009.

[8] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst.*, 30(4):14:1–14:26, 2012.

[9] Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *CNSM*, pages 9–16, 2010.

[10] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing Performance Isolation Across Virtual Machines in Xen. In *Middleware*, pages 342–362, 2006.

[11] N. Huber, F. Brosig, and S. Kounev. Model-based self-adaptive resource allocation in virtualized environments. In *SEAMS*, pages 90–99, 2011.

[12] J. Li, A. C. König, V. R. Narasayya, and S. Chaudhuri. Robust estimation of resource consumption for SQL queries using statistical techniques. *PVLDB*, 5(11):1555–1566, 2012.

[13] H. C. Lim, S. Babu, and J. S. Chase. Automated control for elastic storage. In *ICAC*, pages 1–10, 2010.

[14] B. Mozafari, C. Curino, A. Jindal, and S. Madden. Performance and Resource Modeling in Highly-Concurrent OLTP Workloads. In *SIGMOD*, 2013.

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

[16] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *ICAC*, pages 69–82, 2013.

[17] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *EuroSys*, pages 13–26, 2009.

[18] P. J. Rousseeuw and A. M. Leroy. *Robust Regression and Outlier Detection*. Wiley, 1987.

[19] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *SOCC*, pages 5:1–5:14, 2011.

[20] P. Shivam, A. Demberel, P. Gunda, D. Irwin, L. Grit, A. Yumerefendi, S. Babu, and J. Chase. Automated and on-demand provisioning of virtual machines for database applications. In *SIGMOD*, pages 1079–1081, 2007.

[21] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *SIGMOD*, pages 953–966, 2008.

[22] C. Spearman. The proof and measurement of association between two things. *American J. of Psychology*, 15:72–101, 1904.

[23] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in db2. In *VLDB*, pages 1081–1092, 2006.

[24] A. S. Tanenbaum and D. J. Wetherall. *Computer Networks*. Pearson, 5th edition, 2011.

[25] H. Theil. A rank-invariant method of linear and polynomial regression analysis. *Nederl. Akad. Wetensch.*, 53, 1950.

[26] B. Trushkowsky, P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. The SCADS director: Scaling a distributed storage system under stringent performance requirements. In *FAST*, pages 163–176, 2011.

[27] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.