PerfEnforce: A Dynamic Scaling Engine for Analytics with Performance Guarantees

Jennifer Ortiz[†], Brendan Lee[†], Magdalena Balazinska[†], Joseph L. Hellerstein[‡] and Johannes Gehrke[*]
[†]Department of Computer Science & Engineering,[‡]eScience Institute,[*]Microsoft
University of Washington, Seattle, Washington, USA
{jortiz16, lee33, magda}@cs.washington.edu, jlheller@uw.edu, johannes@microsoft.com

ABSTRACT

We present PerfEnforce, a system that enables performance-oriented service level agreements (SLAs) for data analytics. Given a set of tenants and query-level performance SLAs, we address (i) how to dynamically assign compute resources to queries from each tenant (*query scheduling*), and (ii) how to dynamically resize the multi-tenant service to minimize costs due to compute resources and SLA violation penalties (*resource provisioning*). In addition to its new architecture, PerfEnforce includes a novel approach to resource allocation based on online learning. We find that for query scheduling, online learning outperforms controller and reinforcement learning methods. For resource provisioning, we find that using a simulation-based approach in conjunction with our online learning scheduler leads to cheaper overall costs compared to provisioning based on overall service utilization. In a thorough experimental study on our running data analytics service, we show that PerfEnforce reduces total costs of operating the service by up to 40% compared to a static allocation of resources, and that PerfEnforce can meet even strict tenant SLAs with only 4% percent cost overhead compared to a clairvoyant scheduler.

PVLDB Reference Format:
Jennifer Ortiz, Brendan Lee, Magdalena Balazinska, Joseph L. Hellerstein and Johannes Gehrke. PerfEnforce: A Dynamic Scaling Engine for Analytics with Performance Guarantees. *PVLDB*, 11(2): xxxx-yyyy, 2017.
DOI: https://doi.org/TBD

1. INTRODUCTION

A variety of systems for data analytics are available as cloud services today, including Amazon Elastic MapReduce (EMR) [6], Amazon Redshift [5], Azure's HDInsight [9], Azure Data Lake Analytics [46] and several others. The cloud service providers currently sell availability-based SLAs – a slice of the service with a number of compute resources and a service uptime guarantee [5, 9]. However, users have a hard time determining their resource needs, and they often have to try out many configurations before finding a suitable setup [25].[1]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 44th International Conference on Very Large Data Bases, August 2018, Rio de Janeiro, Brazil.
Proceedings of the VLDB Endowment, Vol. 11, No. 2
Copyright 2017 VLDB Endowment 2150-8097/17/10... $ 10.00.
DOI: https://doi.org/TBD

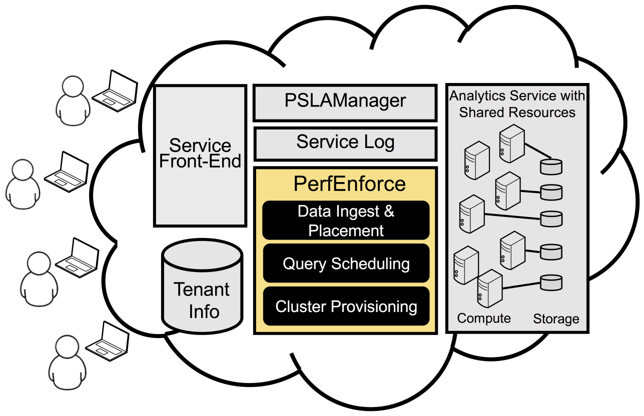


Figure 1: PerfEnforce deployment: PerfEnforce works with an elastically scalable data management system in support of performance-oriented SLAs provided by PSLAManager.

What users really want to purchase, however, are *performance* SLAs — SLAs at the query level [42, 43], where every query has an associated SLA, for example in the form of a query latency. To meet these performance SLAs, the service has to adhere to them, and thus the service provider has to solve the *query scheduling problem*, the problem of dynamically allocating a suitable subset of the existing compute resources for each query to meet its SLA. In addition to solving the query scheduling problem, the service provider may also need to adjust the overall set of compute resources that are associated with the service as the overall load imposed by tenants can grow and shrink. This is a very different problem than query scheduling since it needs to determine the anticipated growth (or shrinkage) of the compute resources of the service based on the number of resources currently in use by all tenants and historical data. We call this problem the *resource provisioning problem.* These problems are hard for several reasons. First, performance SLAs are based on query time estimates, which are often inaccurate [53]. Second, in spite of using VMs and containers to avoid tenant interference, resource availability may change over time which could affect query runtimes. Third, in cloud environments, resource provisioning is relatively fast but it nevertheless adds a visible delay, especially if the service must launch new VMs and find available physical machines to run them. Fourth, ad-hoc analytics renders user workloads less predictable and more variable than periodic, report-generating workloads.

[1]Some systems do not offer any configuration choices but also no performance guarantees, such as Google BigQuery [10].

In this paper, we propose an novel pluggable query scheduling and resource provisioning component called PerfEnforce for cloud data analytics services as shown in Figure 1. PerfEnforce addresses both query scheduling and resource provisioning for data analytics systems such as Myria [52], Spark [8], Impala [34], and EMR [5]. PerfEnforce targets cloud services that follow the Amazon EC2 [3] and Azure HDInsight [9] models, where each user performs her analysis using a conceptually dedicated set of virtual machines (VMs) or containers. Given a service and a set of tenants executing ad-hoc analytical queries, PerfEnforce performs two tasks. First, it schedules these queries in the service by allocating resources from the service (the query scheduling problem). Second, it resizes the service over time based on the overall multi-tenant workload by adding and removing resources (the resource provisioning problem).

Users of the data analytics cloud service can now purchase performance-focused SLAs, which we assume are generated by a separate SLA generator. Although current cloud service providers only give availability-based SLAs, more recent work has focused on generating latency-based SLAs at the query level [42, 55]. The problem is that these SLAs are generated with high uncertainty. To guarantee these performance SLAs, PerfEnforce uses a novel query scheduler that decides the amount of resources to use for each tenant query. Additionally, it reduces costs by adapting resource provisioning algorithms to the setting of an elastically-scalable service. We also show how SLA uncertainty can decrease over time as we learn more about tenant workloads.

In summary, we make the following contributions:

- We develop PerfEnforce, a new system that can be layered on top of a cloud analytics service and which can drive down the costs of performance-oriented SLAs through efficient resource allocation and provisioning. We describe the overall architecture and APIs in Section 2.
- We develop a new query scheduler for multi-tenant data analytics with performance SLAs. Our scheduler is based on an elastic resource allocator that we drive through an online learning algorithm. We compare our approach to feedback control and reinforcement learning (Section 3).
- We adapt the traditional utilization-based technique and develop a simulation-based technique for the service provisioning problem. As a contribution, we evaluate the efficiency of these algorithms in our context of an elastic analytics service with performance guarantees (Section 4).
- Through a thorough experimental evaluation, we find that PerfEnforce is an effective new architectural component in data analytics systems. Query scheduling with our novel online learning approach can help save up to 40% in costs compared to static scheduling and up to 20% compared to a reinforcement learning approach. In addition, our simulation-based resource provisioning approach can save between 7%-45% in costs compared to a traditional utilization-based approach (Section 5).

We discuss related work in Section 6.

2. SYSTEM ARCHITECTURE

In this section, we introduce a prototypical architecture of a cloud data analytics service in which we can deploy PerfEnforce. Figure 1 shows the system components. The front-end for tenants interacts with the back-end service query execution engine, which runs across a variable number of compute nodes and mounts storage nodes as necessary by the queries. The system contains an SLA generator, such as PSLAManager [42] which we describe below and which generates performance SLAs for tenants. The Tenant Info stores various metadata about tenants such as information about their data and the types of SLA they purchased. The service log keeps track of the history of interactions of a tenant with the system, including past queries and their query execution details. These logs serve to improve the accuracy of performance SLAs over time. They are also used by PerfEnforce, the new component that we introduce in this paper, to build models for query scheduling and resource provisioning.

To use the service, a tenant u first loads her data, then purchases a service tier $tier_u$, which specifies the SLA for the tenant, and then proceeds to issue (interactive) queries after starting a query session. As in many cloud services, we make the assumption that the tenant pays a fixed amount (based on her SLA) per time unit; we assume the time is one hour but our techniques can be generalized to work with smaller or larger time units. The tenant thus pays the fixed hourly price of the SLA until the session is terminated. During the session, the tenant submits a sequence of queries. We denote the sequence of queries issued by user u as $\mathcal{W}_u$. This set holds j ordered pairs, where each pair contains a query q_j and its corresponding (promised) SLA time, $t_{sla}(q_j)$: $\mathcal{W}_u = (q_1, t_{sla}(q_1)), \ldots, (q_j, t_{sla}(q_j))$. The PSLAManager associates $t_{sla}(q)$ with each user query q based on the tenant's SLA stored in Tenant Info. As discussed in the introduction, we consider interactive data analysis sessions, where $\mathcal{W}_u$ is not known a priori; without loss of generality, we define a session as a sequence of non-overlapping queries.

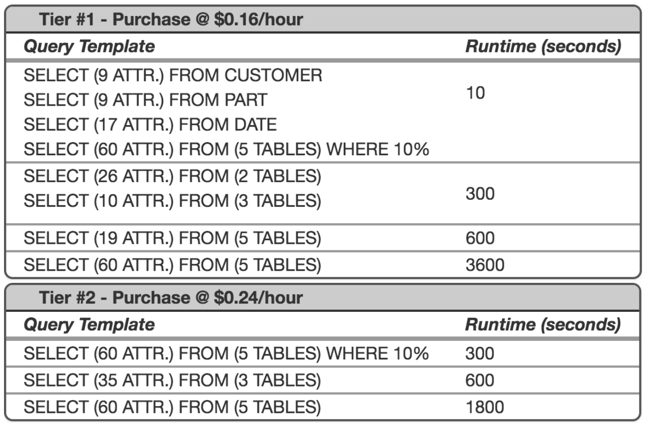
Every time a user submits a query, the service front-end uses the SLA generator to associate a performance SLA time for the query. It then gives the query and its associated SLA time to PerfEnforce, which schedules the query for execution (the query scheduling problem) and resizes the service as needed (the resource provisioning problem).

We now describe each component in more detail.

2.1 PSLAManager: The SLA Generator

PerfEnforce supports the performance guarantees of the SLAs by carefully orchestrating the cloud service's resources. We build on our own performance SLA generator, the PSLAManager system [42], to make our instantiation concrete, although PerfEnforce could work with others. Conceptually, PSLAManager exposes an API with two functions as shown in Table 1. The first method, `GenerateSLA`, takes as input a database schema and statistics associated with a concrete database instance for a tenant, u (we use the term user and tenant interchangeably). It generates an SLA specific to the database instance. For example, a user who wishes to analyze the TPC-H Star Schema Benchmark [41], could see the performance SLA shown in Figure 2. The SLA comprises k performance-oriented service tiers, $\{Tier_1, \ldots, Tier_k\}$. Each tier is defined by a fixed hourly price, along with sets of grouped queries where each group contains a time threshold (shown as "Runtime" in the figure). The time threshold represents the upper bound performance guarantee for its respective group of queries. In this paper, we restrict our attention to select-project-join (SPJ) query templates but the approach is more generally applicable. Each tier represents a performance summary for a specific configuration of the cloud service. A cloud *configuration* represents a set number of resources that can execute a query. We refer to all available cloud configurations in the service as the set $configs$.

All query templates displayed in the SLA are based on the user's dataset. Runtimes shown are predictions from an offline model trained on a separate synthetic dataset. Initially, SLA estimates



Tier #1 - Purchase @ $0.16/hour	
Query Template	***Runtime (seconds)***

SELECT (9 ATTR.) FROM CUSTOMER
SELECT (9 ATTR.) FROM PART
SELECT (17 ATTR.) FROM DATE
SELECT (60 ATTR.) FROM (5 TABLES) WHERE 10%
	10

SELECT (26 ATTR.) FROM (2 TABLES)
SELECT (10 ATTR.) FROM (3 TABLES)
	300

SELECT (19 ATTR.) FROM (5 TABLES)	600
SELECT (60 ATTR.) FROM (5 TABLES)	3600

Tier #2 - Purchase @ $0.24/hour	
Query Template	***Runtime (seconds)***
SELECT (60 ATTR.) FROM (5 TABLES) WHERE 10%	300
SELECT (35 ATTR.) FROM (3 TABLES)	600
SELECT (60 ATTR.) FROM (5 TABLES)	1800

Figure 2: Example two-tier SLA provided by PSLAManager. Each tier represents a performance summary based on a configuration of the database management system. Once the user selects a tier, they can begin the query session.

Function name and parameters	**Returned value**
GenerateSLA $(schema_u, stats_u)$	$\{tier_1, \ldots, tier_k\}$
QuerySLA $(q, tier_u)$	$t_{sla}(q)$

Table 1: SLA Generator API

are likely to be inaccurate. As tenants run queries during their session, we demonstrate in Section 5 how runtime estimates for SLA runtimes can improve over time.

In Tier #1, there are four different time thresholds shown. Naturally, higher runtime guarantees correspond to more complex queries. Tier #2 contains all the performance benefits of Tier #1, but only displays the performance improvements over Tier #1. For example, at the 300 runtime threshold, the Tier #2 configuration is able to run a query that projects more attributes and can join across more tables compared to Tier #1. Although Tier #2 corresponds to a larger cluster size, not all query runtimes improve. In addition, higher tiers come with higher prices as they provide performance guarantees for larger cluster sizes. The number of service tiers shown to a user is configurable.

The query runtimes shown in the SLA depend on statistics about the data, *stats*. These statistics could be provided directly as input by the tenant or, more conveniently, can be collected by the data analytics service as the data is loaded, or by scanning or sampling the user's data. We use the latter approach in our prototype implementation, assuming that the tenant's data is available either because it has already been ingested in the data analytics system or because it is stored in a cloud service such as Amazon S3.

The second method, `QuerySLA`, takes as input a concrete query and a specific service tier previously generated for a user. It returns $t_{sla}(q)$, the SLA runtime for the query at that service tier. PSLAManager uses a model that enables the generation of the above SLAs. We do not consider how SLAs are generated in this paper and instead refer the reader to our PSLAManager paper [42]. Since the generation of these SLAs is not our focus, we assume that they are provided to us. Since PSLAManager contains a model of estimating query costs (to produce the performance SLAs), we will also use this model to jumpstart our query scheduling decisions; we describe this in more details in Section 3.4.

2.2 Analytics Service

All tenants share a set of compute and storage resources; we assume that the service has an associated pool of virtual machines (VMs) or containers. These machines run a resource manager such as YARN [7] and a distributed, shared-nothing query execution engine. Each query submitted by each tenant gets allocated a set of YARN containers across the VMs. When a tenant executes a query, PerfEnforce's query scheduling algorithm determines the number of containers needed to run the query. It then selects that number of containers from the shared pool. In addition, PerfEnforce's resource provisioning algorithm determines when to grow or shrink the pool of shared containers/VMs based on the executing queries. In our experiments, we schedule one container per VM and thus use the terms interchangeably in the paper.

For completeness, we describe the details of our system in this section. In general, our architecture can work with cloud data services that have the ability to elastically scale resources and provide the flexibility to separate compute from storage.

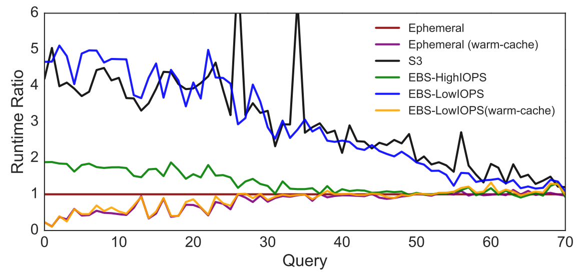


Figure 3: Query Runtimes Compared to Local Storage.

2.2.1 Storage

In Figure 3, we show the median query execution times across three runs for a variety of storage options available on Amazon EC2. The y-axis shows the runtime relative to local storage. Queries on the x-axis are sorted by local storage runtimes in ascending order. The 70 random queries shown are based on a 100SF TPC-H SSB dataset on Myria [52].

Local storage is both cheap and fast, but there is a significant disadvantage to using this storage with elastic scaling. If VMs are added and removed dynamically, the system must constantly migrate data, which is slow. For example, migrating a 100GB dataset from 32 workers (the largest configuration we consider) down to 4 workers takes approximately 40 minutes. An alternative is to replicate data on ephemeral storage throughout the shared pool. If idle VMs are removed from the service, a separate copy of the data could be used to answer the query. The challenge is that, when VMs are added or removed, the system must decide *when* to replicate without causing performance interference. Additionally, when reading data, the system needs to determine which replica to read with minimal impact on other tenants. For these reasons, we do not find ephemeral storage to be a practical option.

As the figure above shows, cheap networked storage such as Amazon S3 or Amazon EBS with a low IOPS setting can slow down queries and lead to high performance variance. Amazon S3's runtimes varied by up to 130% for queries spanning only a few minutes. For EBS-lowIOPS, we observed variance up to 17%. EBS-lowIOPS has the potential of matching the performance of ephemeral storage if the data gets cached in VM memory. Caching, however, is not a reliable solution as the cache might be lost if another tenant uses the VM or if the volume detaches and migrates to a separate VM. Similarly to S3, a shared file system such as Amazon EFS (not graphed) is another available option, but we find that the throughput is considerably lower compared to the other storage options.

EBS-HighIOPS is a remote storage that performs similarly to local storage, while remaining affordable at less than 20% of the cost of a VM. Since this data is persistent regardless of whether we add or remove VMs in our service, we can replicate the data

Function name and parameters	**Returned value**
AddVM()	V_{id}
RemoveVM (V_{id})	void
ExecuteQuery (q_u,$\{V_x, \ldots, V_y\}$)	void

Table 2: Query Execution Engine API. The ExecuteQuery call is blocking and returns when the query completes. Query results are written to disk and retrieved through a separate call if needed.

Function name and parameters	**Returned value**
Initialize ($data(u)$, $configs$)	s_u
Query (s_u, q_j, $t_{sla}(q_j)$)	void
Terminate (s_u)	void

Table 3: PerfEnforce's API.

at ingest time to make scaling efficient. We describe this in more detail in the next section. Thus, we adopt fast networked storage in the service as shown in Figure 1.

2.2.2 *Data Ingest and Placement*

During the initialization of a new query session for a user, PerfEnforce prepares the user data by staging it across multiple EBS volumes. During query execution, it attaches the EBS volumes to different VMs and detaches them as needed. Each EBS volume is attached to a different VM and holds a partition of the data, resulting in a standard shared-nothing configuration. To ingest data quickly in preparation for the query session and to facilitate scaling with minimal interruptions during the query session, PerfEnforce ingests multiple copies of each table from the user's data, $data(u)$. Each copy is uniformly partitioned across a subset of EBS volumes that correspond to one configuration in $configs$. For example, one copy of a table is partitioned across four volumes, a second copy is partitioned across six volumes, a third across eight, etc. To avoid ingesting multiple full copies of each table in sequence for each configuration in $configs$, PerfEnforce uses a technique similar to consistent hashing [30]. Let $Partitions_R = \{p_{r_1}, \ldots, p_{r_m}\}$ represent the partitions of relation, R, where m is the number of EBS volumes in the largest service configuration in $configs$. For example, assume $configs = \{2, 4\}$ and $m = 4$. PerfEnforce first partitions relation R across four volumes as $Partitions_R = \{p_{r_1}, p_{r_2}, p_{r_3}, p_{r_4}\}$. To generate a 2-volume partition, PerfEnforce hash partitions the data from p_{r_3} and stores it on EBS volumes r_1 and r_2 respectively as new partitions $p_{r_{31}}$ and $p_{r_{32}}$. Similarly, p_{r_4} is evenly distributed among workers r_1 and r_2 as $p_{r_{41}}$ and $p_{r_{42}}$. PerfEnforce rewrites queries to use the appropriate set of materialized views for a desired number of VMs. After ingesting larger tables for all $configs$, PerfEnforce also replicates smaller dimension tables on each EBS volume to allow local join execution.

2.2.3 *Query Execution Engine API*

The back-end query execution engine must expose the API shown in Table 2. This API shows the conceptual operations that PerfEnforce requires. Different concrete instantiations are possible. The API includes methods to grow and shrink the number of VMs. It also includes a call to execute a query in parallel across a pre-defined set of VMs. Conceptually, all VMs that execute a query read partitions of the input data and execute the same query plan on those data partitions, reshuffling data during execution as needed.

2.3 PerfEnforce

PerfEnforce exposes an API with three methods as shown in Table 3. The `Initialize` method takes as input the data from the user $data(u)$ and a set of service sizes, $configs$. This method selects a set of virtual machines (VMs) connected to EBS volumes and ingests $data(u)$ for the user as described in Section 2.2.2. After the ingest completes, the method returns a unique session identifier, s_u.

Each call to the method `Query` passes a session identifier as input, the SQL query, q_j, to execute, and the SLA runtime associated with that query. The service front-end uses PSLAManager to obtain this SLA time before invoking the `Query` method. The `Terminate` cleans up all state associated with a query session.

3. DYNAMIC QUERY SCHEDULING

PerfEnforce's objective is to minimize the cost to the cloud provider associated with offering performance SLAs. This cost includes the cost of the VMs in the shared multi-tenant service and the penalties due to SLA violations. To achieve this goal, PerfEnforce performs two tasks: query scheduling and resource provisioning. The scheduling algorithm decides how many VMs to use to execute each query in each session and we present it in this section.

We start by formalizing PerfEnforce's optimization problem.

3.1 Optimization Function

Consider a cloud service operation interval $T = [t_{start}, t_{end}]$. The total operating cost to the cloud during that interval is the cost of the resources used for the service and the cost associated with SLA violations for tenants active during that interval. Thus, PerfEnforce's goal is to minimize the following cost function :

$$\text{cost}(T) = \text{cost}_R(T) + \sum_{u \in U(T)} (\text{penalty}(u)) \qquad (1)$$

where $U(T)$ is the set of all tenants active during time interval, T, and shared resource cost, $cost_R(T)$, is given by:

$$\text{cost}_R(T) = \sum_{t=t_{start}}^{t_{end}-1} \text{cost}_t(\text{resources}) \qquad (2)$$

where $cost_t(resources)$ represents the cost of the shared physical resources for time interval $[t, t+1]$, which depends on the size and the price of individual compute instances.

The SLA penalty, $penalty(u)$, is the amount of money to refund to user u in case there are any SLA violations. There are different ways to specify an SLA and a penalty for SLA violations. In this paper, we use the following simple formulation:

$$\text{penalty}(u) = \mathcal{S}\Big(\frac{1}{|\mathcal{W}_u|}\sum_{q \in \mathcal{W}_u} \max(0, \frac{t_{real}(q) - t_{sla}(q)}{t_{sla}(q)})\Big) * \alpha * p_u \qquad (3)$$

where $\mathcal{W}_u$ is the sequence of queries executed by user u, $t_{real}(q)$ is the real query execution time of query q, $t_{sla}(q)$ is the SLA time of q, p_u is the query session price paid by user u in the absence of SLA violations, and α is a configurable parameter that we vary in our experiments to adjust the cost of SLA penalties compared with VM resource costs. $\mathcal{S}$ is a step function representing the service credit provided to each user. For example, if the system misses the SLA time by an average of 10%, no service credit is provided. If the average percentage is higher, between 10% and 20%, a 20% service credit is given, etc. In short, our step function ensures service credits proportional to the magnitude of the SLA violations. We refer to the output of this function as the *query runtime penalty* since it is independent from p_u.

We use this cost function as an example of a metric that measures the trade-off between reducing query performance penalties and minimizing service expenses. This step function is inspired by real SLAs in cloud services today that incur penalties based on availability outages [9, 49]. Note, no credit is given for queries that complete faster than their SLA times.

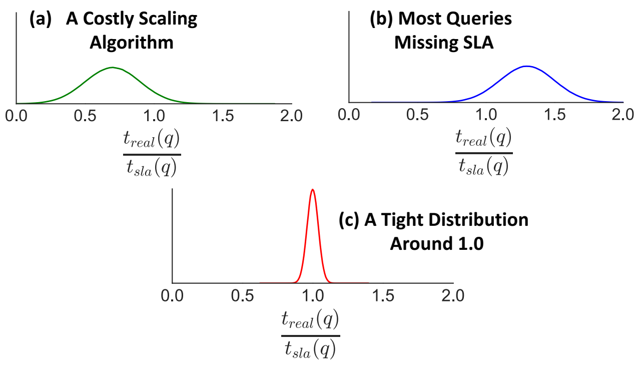


Figure 4: Examples of Distributions of Query Performance Ratios

3.2 Query Scheduling Algorithms

For each query $q \in \mathcal{W}_u$ and for each user u, PerfEnforce's query scheduling algorithm must determine the amount of resources from the shared pool to allocate to the query. SLA tiers initially correspond to a specific set number of physical resources and SLA runtimes are query execution time estimates for each of these configuration options. The query scheduling algorithm could schedule each query on as many VMs as the user purchased through their SLA. Query time estimates, however, are typically inaccurate due to cardinality estimation errors and runtime prediction errors. Changes in cloud conditions can further contribute to estimation errors. To counter these two sources of errors, PerfEnforce may execute queries using more or fewer VMs than initially planned. It does so by using a *scaling* algorithm. It runs a scaling algorithm separately for each tenant. In this section, we discuss both reactive and proactive scaling algorithms implemented in PerfEnforce. All algorithms start by executing queries in the purchased number of VMs or in a randomly selected number of VMs. Reactive algorithms observe errors between the real and SLA runtimes and adjust the number of VMs accordingly for each query. Proactive algorithms, continuously update their knowledge of query execution times for different configurations, and schedule each query based on that knowledge.

3.2.1 Scaling Objective

To minimize resource costs, the scaling algorithm should schedule queries on the smallest possible number of VMs. To avoid SLA penalties, however, it must schedule queries on sufficiently large numbers of VMs to ensure that the real query execution time, $t_{real}(q)$, is below the SLA time, $t_{sla}(q)$. We define the *query performance ratio* as $\frac{t_{real}(q)}{t_{sla}(q)}$ and the goal of the query scheduling algorithm is thus to execute each query in the configuration that yields a performance ratio of 1.0. In practice, if the query scheduling algorithm aims for query performance ratios of X, it will yield a query performance ratio distribution around X as illustrated in Figure 4. To illustrate our point, we plot synthetic Gaussians. Real distributions are noisier. Since we can adjust the mean of the distribution (a.k.a. setpoint), X, the quality of the scheduling algorithm is determined by the tightness of the distribution around X. In other words, if the distribution is wide (large standard deviation σ), then the system is either wasting resources for many queries (Figure 4a) or causes large numbers of SLA violations (Figure 4b). A good query scheduling algorithm should yield a tight distribution as in Figure 4.

3.3 Reactive Scaling Algorithms

We implement a Proportional Integral (PI) controller and a Multi-Armed-Bandit (MAB) as our reactive methods. Both of these techniques have successfully been used in other resource allocation contexts [31, 33, 36, 38].

Proportional Integral Control (PI). Feedback control [27] in general, and PI controllers in particular, are commonly used to regulate a system in order to ensure that it operates at a given reference point. With a PI controller, at each time step, t, the controller produces an actuator value $u(t)$. In our scenario, this is the number of VMs to use to execute the current query. The actuator value, causes the system to produce an output $y(t+1)$ at the next time step. we compute, $y(t)$, as the average query performance ratio over some time window of queries w: $y(t) = \frac{1}{|w|}\sum_{q\in w}\frac{t_{real}(q_j)}{t_{sla}(q_j)}$ where $|w|$ is the number of queries in w. The goal is for the output, $y(t)$, is to be equal to some desired reference output $r(t)$, 1.0 in our setting.

The error $e(t) = y(t) - r(t)$ captures a percent error between the current and desired average runtime ratios. Since the number of VMs to spin up and remove given such a percent error depends on the configuration size, we add that size to the error computation as follows: $e(t) = (y(t) - r(t))u(t)$.

The PI controller, chooses the next number of VMs as a combination of the initial configuration size $u(0)$, the most recently observed error, $e(t)$, and the sum of all accumulated errors $\sum_{x=0}^{t} e(x)$. k_p and k_i are tunable controller parameters, which determine how strongly the controller reacts to recent errors and how much it weighs history:

$$u(t+1) = u(0) + \sum_{x=0}^{t} k_i e(x) + k_p e(t) \tag{4}$$

Multi-Armed Bandits (MAB). In a MAB problem, the system must repeatedly choose among k different options, or *arms*. At each timestep t, the system makes a decision by selecting one of k arms, a_t, and receives a reward, r_t [48]. In our setting, each arm is a configuration from the set $configs$. The arm choice is the decision to schedule the next query in a given configuration size.

The goal is to maximize the total reward across many timesteps. In the bandit setting, the algorithm must learn the reward distributions for different arms through a process of trial and error [11]. At each timestep, the system must thus choose to either select the arm with the highest estimated reward (*exploitation*) or try another arm (*exploration*) with the goal of acquiring more information and maximizing the total reward across all timesteps [48].

To help balance between exploration and exploitation, we use a heuristic known as *Thompson Sampling* [12]. During initialization, we define priors describing the expected reward of each arm. In our setting, we do not make assumptions for each configuration. Instead, we initialize the model for each arm using a uniform distribution, a noninformative prior. At timestep t, the system constructs a posterior distribution for each arm based on observed rewards, $P(\theta|a, r_0, ..., r_{t-1})$, where θ represents the model parameters. For each query submitted, the system samples from a candidate posterior distribution, defined as $\hat{\theta}$. Given that our prior is based on a uniform distribution, we use a t-distribution to represent our posterior. This t-distribution takes the reward mean, variance, and count as input. As the system samples from this posterior, we select the arm with the highest expected reward, $arg\,max_a \mathbb{E}[P(r_t|\hat{\theta}_\alpha)]$.

A limitation of the above techniques is that the configuration size chosen for a new query depends only on the rewards or errors of previous queries. The best choice, however, may also depend on the features of the current query. For example, the original SLA might have under-estimated the execution time only for queries that include a specific join, and thus only those queries need to be scaled up.

3.4 Proactive Scaling Algorithms

To address the above limitation, we consider two other scaling algorithms that both include additional context, x_q, for each incoming query, where x_q is a $D-$dimensional vector of features describing the query, $x_q = (x_{q_1}, ..., x_{q_D})^T$. To generate the feature vector, we use the query optimizer of the back-end query execution engine and include information from the query plans (e.g. number of columns, estimated minimum and maximum costs across workers, estimated number of rows, estimated width, and the number of workers scheduled to run the query). In our approach, the same feature vector can be used for the SLA Generator and PerfEnforce.

Contextual Multi-Arm Bandit (CMAB). This approach is a well-known variant of the multi-armed bandit problem that includes exactly the type of contextual information present in our setting. In a CMAB problem, at each timestep t, the algorithm receives a feature vector, x_q, as input, and uses it to determine the best arm, a_t. CMAB does this by building a model *for each configuration* that predicts the reward in that configuration given a query feature vector. The expected value of the reward for each arm and feature vector thus becomes: $q_\star(a) = \mathbb{E}[r_t|a_t, x_q, \theta]$.

Where θ represents the parameters of the generated model [12]. As with MAB, PerfEnforce uses the Thomson sampling heuristic to balance exploration and exploitation. At each timestep t, PerfEnforce builds a predictive model for each state by computing a bootstrap sample over all previous observations. PerfEnforce selects the action that corresponds to the state with the best predicted reward (i.e., reward closest to 1.0). In our prototype implementation, we use the REPTree model from Weka [24] as used in BanditDB [38]. For the first N queries in a tenant's session, we begin with a "warm-up" phase where we execute queries a small number of times in each configuration to initialize the observations for that configuration.

Online Learning The CMAB technique described above presents two practical challenges. First, it is difficult to determine the number of queries that should be used to initialize the distributions for each state. At least one query must be executed in each state, which can be either unnecessarily expensive or undesirably slow. The overhead especially penalizes short query sessions as early queries undergo larger amounts of exploration. Second, the observations collected are independent for each state. If one configuration suddenly results in slower or faster runtimes, this knowledge does not propagate to other states.

Because of the above limitations, we propose a different algorithm for our setting. With our approach, we build a single model of query execution times with the configuration size as a feature. We start from an offline model built with training data. This can be the same model as the one used by the PSLAManager. As a user executes queries, we always schedule those queries in configuration sizes expected to yield the best performance ratio and use the resulting query execution times to update our global model. With this approach, we do not pay any overhead for an initialization period. Additionally, we schedule queries in configuration sizes different from the one originally purchased *only* if the original model inaccurately estimated query execution times.

Concretely, PerfEnforce learns a linear model to predict the query execution times for all configurations. More complex models are possible but we find a simple linear model to yield good results for the select-project-join queries that we focus on in this paper. With this model, predictions are made by learning the coefficients (a weight vector, w) [11] given the query context, x_q: $y(x_q, w) = \sum_{d=1}^{D} w_d \cdot x_{q_d}$. To initialize the model, PerfEnforce first learns from separate training datasets *before* any user starts any query session.

Importantly, when a user starts her query session, PerfEnforce *continuously updates* the model based on the query execution times measured for the queries in the session. To update the model, PerfEnforce uses stochastic gradient descent. For each data point, it slowly updates the weight vector based on the gradient of a loss function, E: $w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E$ [11]. Where τ represents the nth data point and η represents the learning rate.

In terms of implementation, we use TensorFlow [1] to build and update this model. Since w is first initialized with random values, we first train the model offline on a separate dataset. This dataset is based on the Parallel Data Generation Framework [45]. We generate 4000 queries (500 queries per configuration) and record the features as well as runtimes across the available configurations. Once a tenant begins their query session, the system loads this trained model and uses it to predict the runtimes for each new query. Each tenant begins their session with a copy of this trained model, but feeds new information back to its own version of the model.

4. DYNAMIC PROVISIONING

To reduce operation costs, PerfEnforce includes a resource provisioning component that determines when to grow and shrink the underlying shared physical resources of the service.

4.1 Utilization-based Provisioning

The most common approach to resource provisioning is to maintain a desired resource utilization level. Typically, this means adding (or removing) resources when CPU, I/O, network and memory usage move beyond (or below) set thresholds [4, 9, 23, 47]. In PerfEnforce, we implement a utilization-based algorithm that we use as a baseline to evaluate our other algorithm.

We posit, however, that measuring resource utilization levels directly is not the right approach for PerfEnforce because tenants are allocated resource containers. As such, some tenants might execute I/O intensive workloads while others may run CPU intensive workloads, leading to very different resource utilization levels for various containers. In general, high resource utilization does not necessarily imply a higher demand for resources [16].

In PerfEnforce, we thus develop an algorithm based on utilization, but instead of aiming for a given CPU or I/O utilization goal, we aim for an average *VM utilization* target, Z. The utilization of each machine is measured by the percentage of time it is actively running queries (wall clock time). For our target Z, we aim for an average utilization across all shared VMs, $AvgUtilization$.

To determine the number of machines the system should provision to meet Z, we implement a PI controller where the set point is Z. Our controller is setup as follows: $V_{new} = V_{init} + (e_i * V_{k_i}) + (e_p * V_{k_p})$, where V_{init} is the number of initial resources in the pool, e_i represents the integral term, and e_p represents the proportional term. V_{k_i} and V_{k_p} are tunable parameters representing the integral and proportional gains, respectively. We note that other metrics for system state could be used as well. For example, the system could target a desired percentage of idle machines or a desired tenant query queue length.

The algorithm is shown in Algorithm 1. At each time step, $\mathcal{M}$, we first compute the $AvgUtilization$ of the shared physical resources (line 5). We use the controller to determine the number of nodes to provision based on the relative error between $AvgUtilization$ and the target value, Z. As previously described in Section 3.2, a controller uses parameters such as proportional gain (V_{k_p}) and integral gain (V_{k_i}) to determine how quickly the system should react given the most recent error and the accumulation of prior errors (line 7). In lines 8-19, we either add VMs to the shared pool or remove them, based on the controller's output.

Algorithm 1 Utilization-based Provisioning Algorithm

```
1: Given: $Z$ (utilization target), $V_{k_i}$ (integral gain), $V_{k_p}$ (proportional gain)
2: $V \leftarrow$ set of current nodes
3: $e_i = 0$
4: for each $\mathcal{M}$ time units do
5:     $AvgUtilization \leftarrow \frac{1}{|V|} \sum_{v \in V} Utilization(v)$
6:     $e_p = \frac{AvgUtilization - Z}{Z} \cdot |V|$
7:     $V_{new} = V_{init} + (e_i * V_{k_i}) + (e_p * V_{k_p})$
8:     if $V_{new} - |V| > 0$ then
9:         $numAdd = V_{new} - |V|$
10:        for $a \rightarrow 1, numAdd$ do
11:            $V = V \cup AddVM()$
12:    else if $V_{new} - |V| < 0$ then
13:        $R = 0$
14:        $numRemove = |V| - V_{new}$
15:        for each $v \in V$ do
16:            if $isFree(v)$ and $R < numRemove$ then
17:                $RemoveVM(v)$
18:                $V = V - \{v\}$
19:                $R+ = 1$
20:    $e_i + = e_p$
```

Algorithm 2 Provisioning with User Workload Simulations

```
1: // For a shared resource size, sharedConfigCosts returns of a list of possible costs
2: $sharedConfigCosts : c \rightarrow []$
3: for each $\mathcal{M}$ time units do
4:     // Simulate $s$ times
5:     for $j \rightarrow 1, s$ do
6:         // Sample query candidates for each tenant
7:         for each $u \in U$ do
8:             for $i \rightarrow 1, |recent(\mathcal{W}_u)|$ do
9:                 $sampled_q \leftarrow draw(recent(\mathcal{W}_u))$
10:                $\mathcal{Q}_u \leftarrow \mathcal{Q}_u \cup sampled_q$
11:        // Simulate for each shared resource size in sharedConfigs
12:        for each $c \in sharedConfigs$ do
13:            $cost_R \leftarrow resource\ cost\ based\ on\ c$
14:            $workloads \leftarrow \{\}$
15:            for each $u \in U$ do
16:                $workloads \leftarrow (\mathcal{Q}_u, \lambda_u)$
17:            $runtime_{penalty} = Replay(workloads, c)$
18:            $sharedConfigCosts(c) \leftarrow cost_R + runtime_{penalty}$
19:    $best_{sharedSize} = \min_{\forall c \in sharedConfigs}(\mathbb{E}[sharedConfigCosts(c)])$
20:    // Scale PerfEnforce to $best_{sharedSize}$
```

As shown in line 16, node removals are not aggressive. That is, if a node removal cannot be done (i.e. the node is actively running a query), the node is not removed.

4.2 Simulation-based Provisioning

For a more proactive approach to provisioning, we propose to explicitly consider tenant recent workloads rather than only measure resource utilization. Specifically, we propose to build models of tenant workloads and estimate the smallest number of shared resources to support these modeled workloads. We posit this approach should be more effective than simply looking at utilization, since the latter is tightly coupled with the specific set of executed queries and the query scheduler's resource allocation decisions, which are themselves constrained by the amount of shared resources. To estimate the best number of shared resources to support tenant modeled workloads, we use simulations. This approach is not new and it has recently been used in the "What-If" engine from Tempo[50], where the goal is to simulate the performance of many configurations of the MapReduce Resource Manager. For our work, we aim to understand how such a provisioning algorithm in combination with a learning query scheduler can help make profitable decisions within a multi-tenant service.

In our simulation-based provisioning approach, we model each tenant, u, with a tuple (Q_u, λ_u), where Q_u is a set of queries that the tenant may issue and λ_u is the tenant's average think time between consecutive queries. PerfEnforce learns both values from a recent window of each tenant's query session. Based on these models, PerfEnforce then generates random sessions for each active tenant. To generate a random session, PerfEnforce samples queries from the recent query workload and also samples the think time based on a Poisson distribution. In general, these simulations help PerfEnforce discover whether more resources are necessary for all active tenants or whether nodes should be removed to further save on costs.

Algorithm 2 shows the approach. Every $\mathcal{M}$ time units, PerfEnforce makes a scaling decision as follows: In lines (6-10), a set of query candidates, $\mathcal{Q}_u$, are collected for each tenant, u. Up to $|recent(\mathcal{W}_u)|$ queries are selected, where $recent(\mathcal{W}_u)$ represent the set of queries submitted within the previous window. Queries are sampled with replacement through the $draw()$ function. All selected queries for user u are added to the set $\mathcal{Q}_u$. On line 12, we simulate running the workloads across different shared resources sizes, $sharedConfigs$. For a shared resource size, c, we generate tuples that contain the sampled workload of recent queries, $\mathcal{Q}_u$, along with the think time, λ_u, for each tenant u (lines 15 and 16). On line 17, the $Replay$ function takes these tuples as input and generates random tenant sessions to simulate for c. Once the simulation completes, the $Replay$ function returns a $runtime_{penalty}$ value, which represents the presumed *query runtime penalty* of c. On line 18, we take the sum of this penalty cost and the resource cost, $cost_R$, and append it to the list $sharedConfigCosts$. Once the algorithm iterates through all s simulations, PerfEnforce scales to the best shared resource size by finding the one that is expected to result in the lowest cost, as shown on line 19.

5. EVALUATION

In this section, we evaluate the system end-to-end. We first evaluate how SLAs can improve over time. We then evaluate PerfEnforce's scheduling and provisioning algorithms in a multi-tenant setting and assess PerfEnforce's ability to lower operation costs while providing performance-oriented quality-of-service.

We run PerfEnforce and execute all queries on Amazon EC2 using i2.xlarge (4 ECU, 30 GB Memory) instance types priced at $.15/hr. We consider eight types of query scheduling configurations, each with a different number of compute instances: $\texttt{configs} = \{4, 8, 12, 16, 20, 24, 28, 32\}$. For our underlying shared-nothing, database management system, we use Myria [52]. Myria uses PostgreSQL as its storage subsystem.

To generate each tenant's query sequence, $\mathcal{W}_u$, we alternate between different patterns of queries. For example, one tenant might run small, lightweight queries for a majority of the session before switching to queries with larger joins and higher latencies. Thus, for some random number of queries, k, we define the following three discrete distributions: (1) number of joins, (2) number of projected attributes, and (3) selectivity factor. For the next k queries, we sample from each of these distributions and generate a query that meets all the sampled characteristics. Once k queries are generated, we define new distributions for the next random interval of queries. We use both uniform and skewed (zipfian) distributions.

5.1 Evaluation of SLA Predictions

As described in Section 2.1, the PSLAManager generates SLAs specific to tenant databases. These SLAs provide query performance summaries for different cloud configurations. To predict runtimes for these SLAs, the system uses a model built offline and trained on separate data. Initially, since little is known about each

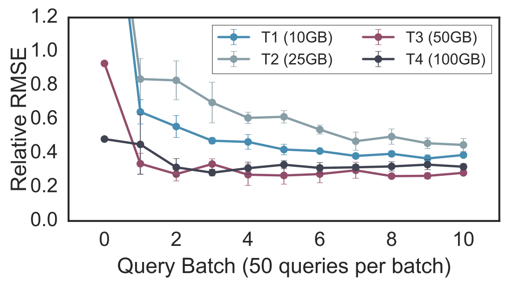


Figure 5: SLA improvements across query sessions

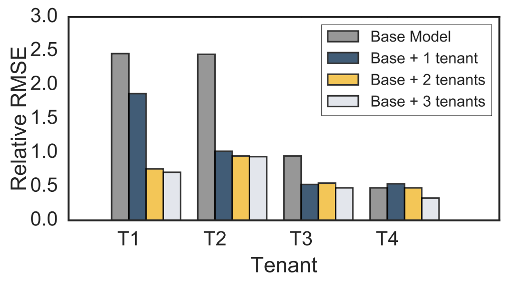


Figure 6: SLA improvements across tenant sessions

tenant's database and query workload, significant mispredictions for these SLAs are likely to occur. As tenants run queries during their session, PSLAManager can incrementally improve the accuracies of the SLAs. This improvement takes two forms: If the same tenant comes back later to purchase a new query session, their SLA will be more accurate because the PSLAManager has a better query runtime model specifically for the tenant (Figure 5). At the same time, as different tenants execute queries, their SLAs can also improve because the PSLAManager has increasingly many data points to learn from (Figure 6).

Figure 5 shows the error of the SLA predictions for four tenants, each with a different star schema and database size : T1 = 10GB, T2 = 25GB, T3 = 50GB, T4= 100GB. As each tenant runs more queries, displayed as query batches on the x-axis, the PSLAManager updates its model based on the service log metadata, and the RRMSEs (relative root mean squared errors) between the real runtimes and the predicted SLA runtimes decreases over time. We compute the error on a sample of queries generated by the PSLAManager for the tenant PSLA. PSLAManager's query generation algorithm systematically creates SPJ queries with specific selectivity factors: .1%, 1%, 10%, 100%. We generate session queries separately using zipfian distributions, which are also SPJ queries but with various distinct selectivity factors. We measure the RRMSE as: $\sqrt{\frac{1}{|\mathcal{W}|}\sum_{q\in\mathcal{W}}(\frac{(t_{real}(q)-t_{sla}(q))}{t_{sla}(q)})^2}$. The prediction errors observed before running an initial batch of queries (Query Batch 0), are highly dependent on the similarity between the tenant's database and the synthetic database used to train our offline base model. In our experiments, while databases differ in their schemas and table sizes, we find that table sizes have the greatest impact on prediction errors. Our offline model is trained on a 100GB synthetic database. We observe a higher initial RRMSE error (approx. 2.4-2.5) for tenants T1 and T2 with the smaller databases. Over time, as each user runs more queries, predictions gradually improve.

Figure 6 shows the SLA improvement across tenants. For the *base model* bar, we show the relative RMSE error for each tenant given a model built only using the 100GB synthetic database. As discussed earlier, T1 and T2 have the highest errors since these databases are most different from the one used to train the base model. For *base model + X tenants*, we show the RRMSE for the updated model that includes queries from "X" tenants *other than the current tenant*. In most cases, initial predictions can improve if queries from a variety of tenants are added to the model. For T4 (100GB), the error slightly increases for *base model + 1 tenant* because the additional tenant generalizes the model to smaller databases, which is not better for T4.

To summarize, SLAs improve over time with our approach, especially when similar tenants use the system or the same tenant purchases the service multiple times.

5.2 Evaluation of Query Schedulers

The goal of each query scheduler is to ensure a tight distribution (small σ) of query performance ratios around a μ close to 1.0. In this section, we evaluate how the different scheduling algorithms perform in the face of different tenant workloads. All tenant workloads are based on the 100GB star schema benchmark.

5.2.1 SLAs Used in Experiments

We evaluate the algorithms using different-quality SLAs, which could correspond, for example, to different model qualities as shown in Section 5.1.

- **Small Gaussian Error (SG SLA)**: This SLA assumes a good prediction model. We generate it by finding the real query execution times at the selected tier and adding a small Gaussian error: $\mathcal{N} = (\sigma = 0.1 * t_{real}(q), \mu = 0)$. This SLA enables us to measure how sensitive the scheduling algorithms are to small errors (or variance) in query runtimes.
- **Positive Gaussian Errors for Large Joins (PLJ SLA)**: We skew the SLA runtimes for specific types of queries only. To generate this SLA, we first collect the real runtimes of queries from the selected tier. We then introduce large positive errors on queries with a large number of joins (> 3 joins) and with a runtime longer than 100 seconds. For each query that meets this criteria, we update the runtime to $t_{real}(q) + |e|$, where e is sampled from a Gaussian distribution, $\mathcal{N} = (\sigma = 0.3 * t_{real}(q), \mu = 0)$. For other queries, we still inject small errors as in the SGE SLA.
- **Negative Gaussian Errors for Large Joins (NLJ SLA)**: This SLA introduces large negative errors, $t_{real}(q) - |e|$, for queries with a large number of joins (> 3 joins) and with runtimes longer than 100 seconds.
- **Initial SLA**: This is the least accurate SLA, where runtimes are generated by an initial offline-trained model. Since mispredictions are likely, this is the most interesting case for PerfEnforce, as it requires the most intricate query scheduling decisions.

5.2.2 Query Performance Distributions

We first evaluate the PI-Control scheduling algorithm on four different SLA types and, in each case, on 10 different, randomly generated, tenant query sessions. We execute the PI controller on each tenant's query session independently and measure the resulting query performance distribution for that tenant. We then compute the average μ and σ across these 10 distributions and plot them in the first row of Figure 7. Because the PI controller has three tunable parameters (k_p, k_i and w), each point in the figure corresponds to one such parameter combination. For each graph, we also plot the average distribution of an Oracle, which always selects the best configuration size for each query. The best parameter combinations are those closest to the Oracle. As the figure shows, for all SLAs, the PI-Control algorithm results in average distributions that are far from the average distribution of the Oracle. These results greatly depend on the tunable configuration parameters and for each figure, a different set of parameters yields the best results. There are no best set of parameters that work across all workloads and SLAs.

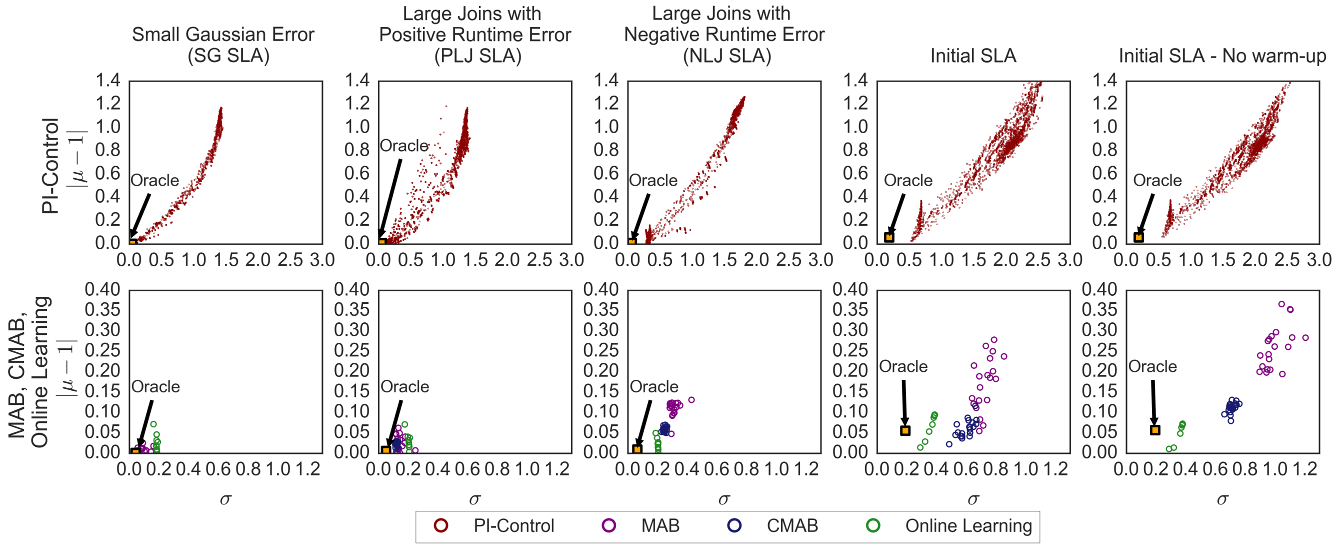


Figure 7: Evaluation of PI-Controller, MAB, CMAB and online learning scheduling algorithms across different SLAs

In the second row of Figure 7, we show the average distributions for MAB, CMAB, and the online learning technique across the same set of SLA types and tenant query sessions. Again, the y-axis represents the distance between μ and 1.0, while the x-axis displays the standard deviation of the query performance distributions. Note, the ranges for the axes are much smaller for these graphs compared to the PI-Controller, which shows that these techniques result in average distributions much closer to the Oracle.

For both bandit techniques, we execute each tenant's query session 20 times due to their variance when sampling. For online learning, we vary the learning parameter, η. In the first 4 columns of the figure, we omit the performance ratios for the first 20 queries for all scaling techniques, since the bandits require an initial "warm-up" phase, where they need to try each configuration at least two times. These figures thus show the best performance for the bandit algorithms, which is their performance once a fair amount of exploration has happened.

For the SG SLA, the bandit techniques result in average distributions nearly identical to the Oracle. Since both techniques rely on learning a distribution of query performance ratios per configuration, they quickly find the optimal configuration during the warm-up phase and select this configuration for a majority of the queries. Since the online learning technique is directly predicting the runtimes for each query, the prediction errors result in average distributions that are slightly farther away from the Oracle. For the PLJ SLA, all techniques perform similarly as most of the runtimes are meet at configurations that are close to the purchased tier. In contrast, online learning outperforms both bandit-based methods for the NLJ and Initial SLAs. The NLJ SLA underestimates runtimes, which requires the schedulers to accurately choose across a wider set of configuration options. For these more difficult cases, context is critical as the scheduling algorithm must make different decisions for different queries. There is no single best configuration. As a result, CMAB and the online learning approach both outperform the simpler MAB scheduler. Online learning further outperforms CMAB because this technique is able to quickly learn the performance correlations between configurations, which is crucial for the initial SLA as it requires scaling for each query.

The final column shows the average performance ratio distributions when using the initial SLA and including the queries in the warm-up period. As the figure shows, the online learning technique significantly outperform the bandit-based methods because it has the extra benefit of starting to learn from the offline model and learning more quickly because it learns a single model for all configurations.

These results show that the PI controller is ill-suited to our problem and we do not consider it further. Our approach requires contextual decisions, which differ between queries. Thus the CMAB and online learning yield the best results. The online learning approach has the added advantages of starting from the offline model and learning a single model, which speeds up its convergence.

5.2.3 *Adapting to Changing Conditions*

We now evaluate how query scheduling algorithms can adapt to changing system or workload conditions. Recall, the goal of these query schedulers is to ensure a query performance ratio distribution close to 1.0.

To experiment with changing system conditions, we first generate a query sequence by selecting one query and running it repeatedly several times. Each time we run this query, we record the query performance ratio. Once we reach the 250th iteration, we increase the query's runtime by 25% (essentially slowing down the system) for the rest of the session, running up to 1000 iterations. A good query scheduling algorithm will react to this slowdown by increasing the number of workers to achieve a query performance ratio close to 1.0. Table 4 shows the mean (μ) and standard deviation (σ) of the query performance ratios for each scaling technique during this query session. We also include the Oracle scheduler as a comparison. The online learning technique closely matches the mean and standard deviation of the Oracle. In contrast, the MAB technique never converges to the best configuration once the slow down occurs, as it adheres to the knowledge it obtains during the beginning of the session. Similarly, for CMAB, the model never fully converges to the best configuration within 1000 queries. This is one disadvantage of having independent models per configuration. During the "warm-up" phase, the CMAB technique initializes its knowledge about each state. Once the system reaches the 250th query, CMAB's belief about each state becomes obsolete. Although sampling helps ensure exploration (allowing the model to revisit other states), it takes several observations before the models can reflect the new runtimes.

We also measure the effect of changing query workloads during a query session. We use the Initial SLA model in this experiment. Table 5 shows mean and standard deviation for each technique. Dur-

	MAB	CMAB	Online Learning	Oracle
μ	1.1368	1.1244	1.0161	1.0015
σ	0.1680	0.0871	0.0522	0.0008

Table 4: Query ratio distributions when slowing down performance

	MAB	CMAB	Online Learning	Oracle
μ	1.4353	1.1421	1.1656	1.1563
σ	0.5898	0.2671	0.2372	0.2214

Table 5: Query ratio distributions with changing workloads

ing the first half of this session, we run queries that have fewer than two joins and selection conditions. For the second half of the query session, we switch the workload to one that has large joins with no additional selection predicates. The Initial SLA model happens to underpredict the execution times for queries with small numbers of joins. Hence, the mean query performance ratio of the Oracle scheduler is slightly above 1.0, since the runtimes for these queries are impossible to meet even with a configuration with the maximum number of VMs. Unlike the MAB technique, the CMAB and Online Learning models benefit from taking into account the query's context to make predictions, as the resulting mean and variance are comparable to the Oracle.

In general, the online learning and CMAB techniques result in similar distributions when considering changing system conditions. The online learning approach, however, converges faster in the case of performance slowdowns.

5.3 Evaluation of Provisioning Algorithms

Each provisioning technique works in combination with a query scheduler: As PerfEnforce schedules queries for each tenant, the provisioning technique adds or removes nodes to or from the shared cluster to minimize cost. We first evaluate each provisioning algorithm in combination with the Oracle query scheduler to ensure that query runtime penalties are not a side-effect of the query scheduler's mispredictions.

We launch each multi-tenant session based on the following *session parameters*:

- U_{init} : Initial number of tenants in the session
- V_{init} : Initial number of virtual machines
- $\lambda_{arrival}$: Average time between new tenants
- $\lambda_{thinktime}$: Average tenant think time
- $\lambda_{terminate}$: Average tenant session duration
- $\mathcal{M}$: Provisioning monitoring time interval

We introduce up to 100 tenants in a session. We sample arrival times, think times, and session durations from Poisson distributions defined by their corresponding parameters $\lambda_{arrival}$, $\lambda_{thinktime}$ and $\lambda_{terminate}$. PerfEnforce always keeps at least a minimum of 4 machines launched at all times, to ensure that there are enough machines available to execute queries. Each provisioning algorithm monitors the shared resources and tenants for $\mathcal{M}$ minutes before adding or removing virtual machines from the pool. Our step function $\mathcal{S}$ provides no service credit if the system misses the runtime by 10%. Further, for each 20% increment and given a threshold from x% to y%, we increase the credit to y%.

As described in Section 2.2, each tenant gets allocated a set of containers across the VMs. We schedule one container (running a Myria process) per virtual machine, but configurations with multiple containers per VM are also possible. For each query, the system assigns the tenant's EBS volumes to a set of virtual machines in the pool. After the query completes, the volumes are detached from the VMs, making them available to other tenants. We find it takes approximately 4 seconds to mount a volume to a VM. Detaching takes approximately 11 seconds. We include these delays in the following experiments.

5.3.1 *Utilization and Simulation-based Provisioning*

We compare the multi-tenant session costs when provisioning VMs using either the utilization-based or simulation-based approaches. We launch 100 virtual machines with 10 initial tenants and set the provisioning monitoring time to $20min$. Figure 8 shows the results and the other experimental parameters. For different average utilizations, Z, we show the results for the best parameter values V_{k_p} and V_{k_i}. The y-axis shows the cost per time unit, while the x-axis shows the value of the α parameter. Recall from Equation 3 that we define α as a tunable parameter that amplifies the weight of the *query runtime penalty* term compared with the shared resource cost. The hash pattern in each bar represents the proportion of the cost due to $Cost_R$, the cost of resources. Other costs come from SLA violations. As expected, if α is low, it is more cost effective to maintain a high utilization to cut on resource costs, while lower utilizations help cut SLA violation penalties when α is large. There is no single best utilization level. The error bars represent the variance of the cost, as we run each experiment 10 times. Costs vary for each trial due to think times, tenant arrival times, and the session termination times as they are all sampled from a Poisson distribution. For each type of SLA, we also show the costs for the simulation-based approach. Simulation-based provisioning has the double-benefit of avoiding any tuning and more cost effectively provisioning shared resources compared to the utilization-based approach. We do not show the PLJ SLA results due to space constraints, but the results are similar to the NLJ SLA.

A benefit of the simulation-based approach is that it optimizes based on the most recent tenant behavior. As described in our experimental setup, tenant workloads are based on "zipfian" distributions, where specific query characteristics are favored when generating segments of these workloads. We now evaluate the simulation-based approach in the setting where tenant workloads follow a "uniform" distribution to determine the number of joins and other query parameters. Figure 9 shows the costs for both the zipfian and uniform workloads with $\alpha = 1$ based on simulation and utilization ($Z = .25$) provisioning. We fix α to 1.0 to weigh query runtime penalties more heavily in our cost function. The costs for uniform distributions are generally higher, as these workloads sporadically require many more resources than those made available. Nevertheless, even with such workloads, the simulation-based technique provides cost improvements ranging from 5%-17% when compared to the utilization-based approach.

5.3.2 *Combining Scheduling and Provisioning*

We now evaluate the performance of simulation-based provisioning in conjunction with various query scheduling algorithms on the initial SLA. Figure 10 shows the results. In this experiment, we vary alpha (x-axis) and measure the total cost compared with an Oracle query scheduler (y-axis). As a baseline, we also include a naive query scheduler, $static$, which simply schedules each query on the configuration initially purchased by the user. We also include utilization-based provisioning at $Z = .25$ (using online learning as the query scheduler). We still initialize the session with 10 tenants, but we start with a larger shared cluster of size 320, allowing enough room to have each initial tenant schedule queries on up to 32 virtual machines. In this experiment, since we also include CMAB, we extend the session times to 180 to ensure the algorithm has more time to operate in steady state (beyond the warm-up phase).

Overall, simulation-based provisioning continues to outperform the utilization-based approach even with a less perfect scheduler. Even when penalties are high, simulation-based provisioning reduces costs by 11% and more for lower penalties. Additionally,

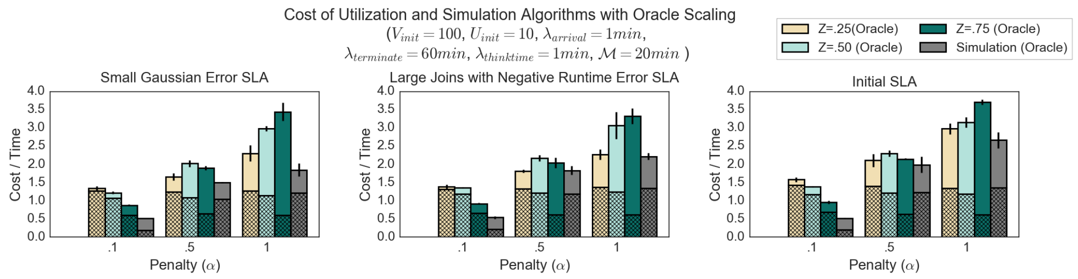


Figure 8: Comparing utilization-based and simulation-based provisioning in conjunction with an Oracle query scheduler. The hash pattern in each bar represents the proportion of the cost due to $Cost_R$, the cost of resources.

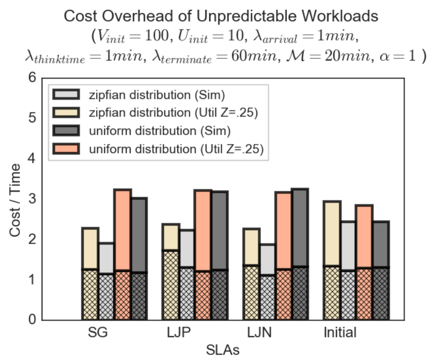


Figure 9: Measuring the overhead of unpredictable workloads

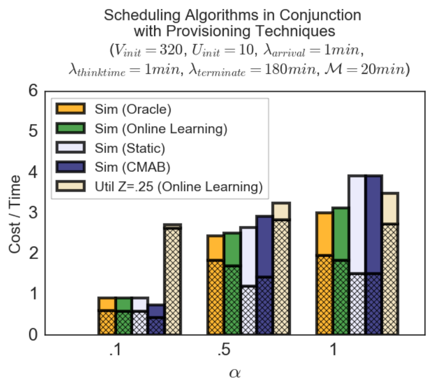


Figure 10: Comparing the costs of query scheduling in conjunction with provisioning

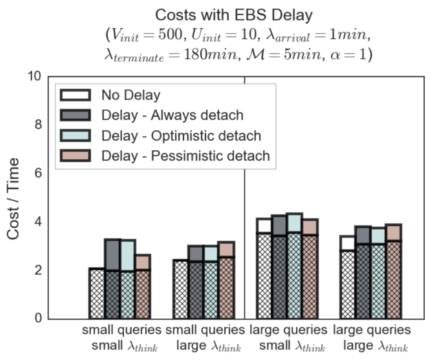


Figure 11: Overhead in costs due to EBS delays and benefits of pessimistic detach heuristic.

the online learning-based scheduler yields similar costs to the Oracle scheduler (a 4% overhead). As expected, it significantly outperforms the static scheduler and CMAB when SLA penalties are expensive, with 20% cost savings. CMAB does worse because it causes more SLA violations. For small α, the CMAB approaches result in costs lower than even the Oracle scheduler. This is because the CMAB's warm-up phase initially schedules queries on all available configurations (even small configurations), which then causes the simulation approach to provision less resources. Throughout the session, resources are not added back in due to the low α value.

5.3.3 *Impact of EBS Delays*

We now evaluate how remote network volumes can impact costs, and the potential benefits of a heuristic, which keeps tenants attached to VMs when their queries are short. We generate four workloads for all tenants: Small queries (< 50 sec.) vs large queries (> 200 sec.) and either short $\lambda_{think} = 5s$ or long $= 60s$ think times. Intuitively, EBS delays will predominately affect query workloads with smaller runtimes, as our cost function penalizes relative runtime errors. If query runtimes are high, the time to attach/detach is a smaller overhead. Additionally, $\lambda_{thinktime}$ is also a factor to consider. If the think times are long, it provides sufficient amount of time for EBS volumes to successfully detach from VMs once a query has completed. Short think times that are less than the detach time can have an adverse effect. 5 seconds is considerably less than the time it takes to detach a volume.

In Figure 11, we show the total cost for all four combinations of short/long queries with small/long think times (x-axis). To make sure we only observe the overhead of attaching and detaching volumes, we use an Oracle scheduler. The "No Delay" bars show the cost assuming that the attaching/detaching actions are instantaneous. The "Delay" bars show the costs with the delays – this is the delay overhead we included in all prior experiments.

As a heuristic to help minimize cost in the presence of these storage delays, we introduce the notion of *detachable tenants*. If a tenant is running long queries, the remote storage delay imposes less overhead in query runtime penalties compared to a tenant running a series of short queries. This is especially true if α is high. Thus, there is a higher incentive for the system to detach the remote volumes from the VMs of those tenants after running each query. To find detachable tenants, we look for the following condition: $mean_{runtime} > \gamma * EBS_{delay}$, where $mean_{runtime}$ represents the average latency of their query workload and γ is a tunable parameter. In our experiments, we select gamma such that only a small percentage of tenants are kept attached. We compare two variants: one that more aggressively detaches tenants (optimistic, $\gamma = 1$ for the short queries experiment and 10 for the long queries experiment) and one that is more conservative and keeps tenants attached (pessimistic, $\gamma = 3$ and 15). For small queries with small think times, there is a 36% increase in costs with EBS delays. Our optimization for keeping tenants attached decreases this overhead to 16%. If the think times are long for short queries (providing sufficient time to detach volumes), keeping tenants attached to resources actually increases the resource cost. For large queries, there is less overhead between "No Delay" and "Delay". Our optimizations end up minimally reducing the cost or minimally increasing the resource cost. In general, the pessimistic approach, which tends to keep volumes attached yields the lowest costs on average.

5.3.4 *Cost of Resources vs QoS*

Finally, we assess PerfEnforce's overall ability to lower resource costs while achieving a good quality of service. In Figure 12, we run 100 tenant sessions across different settings. On the y-axis, we show the cost of resources and on the x-axis we show the query runtime penalty as defined in Equation 3, representing the QoS. The goal is achieve low query runtime penalties at a low cost.

For the "Dedicated" scenario, we run each tenant session in an isolated fixed set of resources. This is the setting currently used

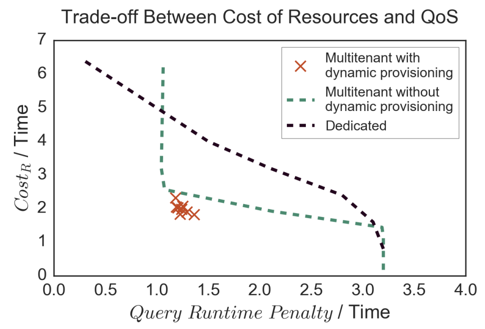


Figure 12: Cost of consolidating compared to dedicated

in cloud services such as Amazon Elastic MapReduce. There is no query scheduler. In the graph, the trend shows the trade-off between costs and query runtime penalties as we vary the number of fixed resources. As expected, if less is spent on resources, the query runtime penalty also increases. We compare this trend to the "Multitenant without dynamic provisioning" approach in which all tenants share a pool of fixed resources without provisioning. The system schedules queries for each tenant based on the online learning approach. Compared to the dedicated scenario, this setup is usually more cost effective. If the cluster is too small, QoS decreases. Once the cluster is sufficiently large, the system provides a good QoS. We also show the resource and QoS trade-off for the "Multitenant with dynamic provisioning" setup. Here, we also vary the number of initial virtual machines we introduce in the session. As the figure shows, this approach effectively provisions the necessary number of virtual machines in a multi-tenant environment regardless of the initial number of VMs. For the same QoS, this setup saves up to 40% in costs compared to static allocation. It automatically achieves the ideal point of good QoS at a low cost.

5.4 Evaluation Summary

We find that an online learning approach can serve as an adaptive scaling algorithm whose predictions lead to cost effective query scheduling decisions. We also find that a simulation-based approach to resource provisioning can adjust the number of shared resources without any parameter tuning and only requires learning about recent tenant behavior. Finally, we observe that an online query scheduler working in conjunction with a simulation-based provisioning approach effectively achieves high quality of service at a low cost in many different configurations. Compared with static single-tenant configurations or multi-tenant configurations without dynamic provisioning, our approach significantly improves QoS for a given cost or lowers costs for a given QoS level.

6. RELATED WORK

Elastic Scaling for Performance Guarantees Performance guarantees have traditionally been the focus of real-time database systems [29], where the goal is to schedule queries in a fixed-size cluster to ensure they meet their deadlines. More recently, dynamic provisioning and admission control methods have enabled OLAP and OLTP systems to make profitable choices with respect to performance guarantees [14, 13, 55], possibly postponing or even simply rejecting queries. Work by Das et. al [16], focuses on using system telemetry and derived characteristics to determine whether users are *likely* to require more resources in the near future. In contrast to our work, they focus on scaling up containers within a single node, whereas our goal is to scale out the number of machines required for each query. Ernest [51], CherryPick [2] and CloudScale's [47] goal is to also find good configurations for analytical workloads, but they require representative workloads or depend on repeating tenant usage patterns. Morpheus [28] also elastically scales to meet SLOs, but relies on periodic jobs.

Several systems have studied performance SLAs through dynamic resource allocation, including storage systems that use feedback control [33], and systems such as TIRAMOLA [31] and BanditDB [38] which leverage reinforcement learning techniques to make optimal decisions. In addition, many other existing solutions leverage decisions based on meeting resource utilization goal [15, 18, 21, 40, 49, 56]. The Tempo [50] system focuses on simulating the performance of many configurations of the MapReduce Resource Manager to meet a global system objective. Their work is in the context of MapReduce systems, where jobs can be preempted to allow tenants with higher priority to complete first.

Multi-Tenant Workload Consolidation An active area of research in multi-tenant cloud DBMS systems is *tenant packing* [19, 37, 35]. Related work addresses this by mitigating bad tenant packing by either finding a good initial tenant placement strategy or dynamically migrating tenants [18, 19, 32, 35, 49, 54]. In contrast, our architecture has each tenant's dataset attach to a container, allowing each tenant to process the data in a more isolated environment. Quasar [18], a system based on Paragon [17], focuses the combination between resource allocation and resource assignment to provide performance guarantees across multiple tenants. Again, finding a good tenant placement strategy is not the focus of our work. In addition, we focus on algorithms that help determine when to launch or turn off machines.

Query Runtime Prediction Previous work has relied on classification and regression techniques to determine whether a query will miss or meet a deadline [55], building gray-box performance models [22], using historical traces of previous workloads [20], using a broad set of benchmarks to profile resources [57], or running smaller samples of the workload with a low overhead [51]. Work by Herodotou et. al. [25], assumes a previously profiled workload from the user in order to predict the runtime of that program against different sized clusters. Work by Jalaparti et. al. [26] focuses on generating resource combinations given performance goals from the user. Instead of building a white-box or analytical model, we focus on using a model that does not require an extensive understanding of a single system. We also focus on interactive, ad-hoc queries for which there are no prior profiles.

Provisioning In terms of resource provisioning, some rely on machine learning techniques such as the hill-climbing approach seen in Marcus et. al. [38], which allows machines to learn an optimal time to wait before they shut down. Neural networks for dynamic allocation [39] or dynamic provisioning [44] have also been used, but have distinct goals. One focuses on allocating resources with minimal use of electrical power while the other assumes an OLTP workload with predictable workloads.

7. CONCLUSION

PerfEnforce is a system designed to reduce the cost of performance SLAs for cloud data analytics. Given a set of tenants with performance SLAs, PerfEnforce provisions a shared cluster of VMs and schedules tenant queries in a way that minimizes operating costs for the cloud. PerfEnforce's query scheduling algorithm uses online learning while its provisioning algorithm models and simulates tenant behavior. We find that PerfEnforce's core algorithms outperform other state-of-the-art methods including a PI controller, bandit-based methods, and utilization-based resource provisioning. Importantly, compared with dedicating resources to tenants, as is done in many cloud analytics systems today, PerfEnforce dramatically improves quality-of-service while keeping costs low.

8. REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, 2017. USENIX Association.

[3] Amazon EC2. `http://aws.amazon.com/ec2/`.

[4] Amazon RDS. `http://aws.amazon.com/rds/`.

[5] Amazon AWS. `http://aws.amazon.com/`.

[6] Amazon Elastic MapReduce (EMR). `http://aws.amazon.com/elasticmapreduce/`.

[7] Apache yarn. `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`.

[8] Apache Spark: Lightnight-fast cluster computing. `http://spark.apache.org/`.

[9] Microsoft Azure. `http://azure.microsoft.com/en-us/`.

[10] Google BigQuery. `https://developers.google.com/bigquery/`.

[11] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[12] O. Chapelle and L. Li. An empirical evaluation of thompson sampling. In *Advances in Neural Information Processing Systems 24*. 2011.

[13] Y. Chi et al. SLA-tree: a framework for efficiently supporting sla-based decisions in cloud computing. In *Proc. of the EDBT Conf.*, pages 129–140, 2011.

[14] Y. Chi, H. J. Moon, and H. Hacigümüs. iCBS: Incremental costbased scheduling under piecewise linear SLAs. *PVLDB*, 4(9):563–574, 2011.

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

[16] S. Das, F. Li, V. R. Narasayya, and A. C. König. Automated demand-driven resource scaling in relational database-as-a-service. In *Proceedings of the 2016 International Conference on Management of Data*, Proc. of the ACM SIGMOD International Conference on Management of Data, pages 1923–1934, New York, NY, USA, 2016. ACM.

[17] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. *SIGPLAN Not.*, 48(4):77–88, Mar. 2013.

[18] C. Delimitrou and C. 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, 2014.

[19] A. J. Elmore, S. Das, A. Pucher, D. Agrawal, A. El Abbadi, and X. Yan. Characterizing tenant behavior for placement and crisis mitigation in multitenant DBMSs. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 517–528, 2013.

[20] A. D. Ferguson, P. Bodík, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, pages 99–112, 2012.

[21] M. Fu, A. Agrawal, A. Floratou, B. Graham, A. Jorgensen, M. Li, N. Lu, K. Ramasamy, S. Rao, and C. Wang. Twitter Heron: Towards extensible streaming engines. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1165–1172, 2017.

[22] A. Gandhi, P. Dube, A. Kochut, L. Zhang, and S. Thota. Autoscaling for hadoop clusters. In *IC2E 2016*.

[23] Z. Gong, X. Gu, and J. Wilkes. PRESS: Predictive elastic resource scaling for cloud systems. In *6th IEEE/IFIP International Conference on Network and Service Management (CNSM 2010)*, Niagara Falls, Canada, 2010.

[24] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. June 2009.

[25] H. Herodotou et al. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proc. of the Second SoCC Conf.*, page 18, 2011.

[26] V. Jalaparti et al. Bridging the tenant-provider gap in cloud services. In *Proc. of the 3rd ACM Symp. on Cloud Computing*, page 10, 2012.

[27] P. K. Janert. *Feedback Control for Computer Systems*. O'Reilly Media, Inc., 2013.

[28] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. n. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated slos for enterprise clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 117–134, Berkeley, CA, USA, 2016. USENIX Association.

[29] B. Kao et al. Advances in real-time systems. chapter An Overview of Real-time Database Systems, pages 463–486. Prentice-Hall, Inc., 1995.

[30] D. Karger et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. STOC, pages 654–663, 1997.

[31] I. Konstantinou et al. TIRAMOLA: elastic nosql provisioning through a cloud management platform. In *Proc. of the SIGMOD Conf.*, pages 725–728, 2012.

[32] W. Lang, S. Shankar, J. M. Patel, and A. Kalhan. Towards multi-tenant performance SLOs. *IEEE Transactions on Knowledge and Data Engineering*, 26(6):1447–1463, 2014.

[33] H. Lim et al. Automated control for elastic storage. In *ICAC*, pages 1–10, 2010.

[34] T. Liu and M. Martonosi. Impala: A middleware system for managing autonomic, parallel sensor systems. *SIGPLAN Not.*, 2003.

[35] Z. Liu, H. Hacigümüs, H. J. Moon, Y. Chi, and W.-P. Hsiung. PMAX: Tenant placement in multitenant databases for profit maximization. In *Proceedings of the 16th International*

Conference on Extending Database Technology, EDBT '13, New York, NY, USA, 2013. ACM.

[36] K. Lolos, I. Konstantinou, V. Kantere, and N. Koziris. Adaptive state space partitioning of markov decision processes for elastic resource management. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 191–194, 2017.

[37] H. A. Mahmoud, H. J. Moon, Y. Chi, H. Hacigümüs, D. Agrawal, and A. El Abbadi. CloudOptimizer: multi-tenancy for I/O-bound OLAP workloads. In *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 77–88, 2013.

[38] R. Marcus and O. Papaemmanouil. Releasing cloud databases for the chains of performance prediction models. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[39] D. Minarolli and B. Freisleben. Distributed resource allocation to virtual machines via artificial neural networks. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 490–499, 2014.

[40] V. R. Narasayya, S. Das, M. Syamala, S. Chaudhuri, F. Li, and H. Park. A demonstration of SQLVM: performance isolation in multi-tenant relational database-as-a-service. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1077–1080, 2013.

[41] P. O'Neil, E. O'Neil, and X. Chen. Star schema benchmark. `http://www.cs.umb.edu/~poneil/StarSchemaB.PDF`.

[42] J. Ortiz, V. T. de Almeida, and M. Balazinska. Changing the face of database cloud services with personalized service level agreements. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[43] O. Papaemmanouil. Supporting extensible performance SLAs for cloud databases. In *Proc. of the 28th ICDE Conf.*, pages 123–126, 2012.

[44] X. Qiu, M. Hedwig, and D. Neumann. *SLA Based Dynamic Provisioning of Cloud Resource in OLTP Systems*, pages 302–310. 2012.

[45] T. Rabl, M. Frank, H. M. Sergieh, and H. Kosch. A data generator for cloud-scale benchmarking. In *TPCTC'10*, pages 41–56.

[46] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos, N. Sharman, Z. Xu, Y. Barakat, C. Douglas, R. Draves, S. S. Naidu, S. Shastry, A. Sikaria, S. Sun, and R. Venkatesan. Azure Data Lake Store: A hyperscale distributed file service for big data analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 51–63, 2017.

[47] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*, page 5, 2011.

[48] R. S. Sutton and A. G. Barto. Reinforcement learning I: Introduction, 2016.

[49] R. Taft, W. Lang, J. Duggan, A. J. Elmore, M. Stonebraker, and D. DeWitt. STeP: Scalable tenant placement for managing database-as-a-service deployments. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, 2016.

[50] Z. Tan and S. Babu. Tempo: Robust and self-tuning resource management in multi-tenant parallel databases. *Proc. VLDB Endow.*, 9(10):720–731, June 2016.

[51] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, Santa Clara, CA, 2016. USENIX Association.

[52] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, D. Moritz, B. Myers, J. Ortiz, D. Suciu, A. Whitaker, and S. Xu. The Myria big data management and analytics system and cloud services. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research*, 2017.

[53] Y. Wang, A. Meliou, and G. Miklau. Lifting the haze off the cloud: A consumer-centric market for database computation in the cloud. *Proc. VLDB Endow.*, 10(4):373–384, Nov. 2016.

[54] P. Wong, Z. He, and E. Lo. Parallel analytics as a service. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 25–36, 2013.

[55] P. Xiong et al. ActiveSLA: a profit-oriented admission control framework for database-as-a-service providers. In *Proc. of the Second SoCC Conf.*, page 15, 2011.

[56] P. Xiong et al. SmartSLA: Cost-sensitive management of virtualized resources for CPU-bound database services. In *IEEE Transactions on Parallel and Distributed Systems*, pages 1441–1451, 2015.

[57] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 452–465, New York, NY, USA, 2017. ACM.