SLAOrchestrator: Reducing the Cost of Performance SLAs for Cloud Data Analytics (Tech Report)

Jennifer Ortiz[†], Brendan Lee[†], Magdalena Balazinska[†], Johannes Gehrke[*] and Joseph L. Hellerstein[‡]
[†] University of Washington Department of Computer Science & Engineering,[*]Microsoft,[‡]eScience Institute

Abstract

SLAOrchestrator is a new system designed to reduce the price increases necessary to support performance SLAs in cloud analytics systems. SLAOrchestrator is designed for SLAs that guarantee per-query execution times. Its core architecture consists of a double learning loop that improves both SLAs and resource management over time. It further utilizes an efficient combination of elastic query scheduling and multi-tenant resource provisioning algorithms to reduce the costs of performance guarantees.

1 Introduction

A variety of shared-nothing systems for data analytics are available as cloud services today, including Amazon Elastic MapReduce (EMR) [5], Amazon Redshift [4], Azure's HDInsight [8], and Azure Data Lake Analytics [48]. When using those systems, users upload their data to the cloud and issue queries on that data. Queries can include relational operators and various user-defined computations. A key challenge with these services, however, is that users must decide on a desired configuration: how many *service instances* they want to pay for and how powerful these instances should be.

The service configuration dramatically impacts price [2] and performance [55], yet it is known to be very difficult for users to select correctly [25]. Since users do not know what configuration to purchase, one approach is to offer performance-based service level agreements (SLAs), where the system promises to meet a given per-query latency or pay a penalty [43, 44].

Previous research has addressed the challenge of selecting and enforcing SLAs in various ways. One line of work assumes each tenant fits on a single server and the challenge is to pack tenants on a restricted set of servers [18, 35, 49], migrating tenants as needed [17], ordering queries for execution [12, 38], controlling admission [58, 44], and dispatching queries to servers [11, 39]. Other approaches assume the workload is known and require profile runs of queries, possibly restricted to processing samples [55, 26, 19, 24]. Knowledge of the workload and profile runs are reasonable assumptions in a transaction-processing system with a fixed set of stored procedures or in an analytics system that runs predefined reports, but not for ad-hoc analytical workloads.

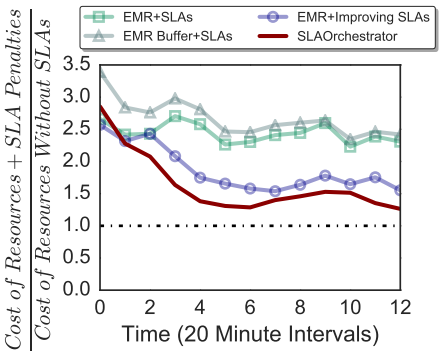


Figure 1: **A time-changing set of tenants executes ad-hoc, analytical queries subject to performance SLAs. Static resource allocation (EMR+SLAs), even with a buffer (EMR+SLA+Buffer) leads to large cost increases. Our improving SLAs (EMR+Improving SLAs), especially with multi-tenancy and other optimizations (SLAOrchestrator), bring costs down.**

Another line of work focuses purely on enforcing SLAs, assuming that SLAs are pre-defined [12, 11, 58]. SLA runtimes are artificially generated by, for example, offering a performance guarantee 10x the true latency [12], or by setting SLAs to be the performance of past executions [30]. Without the right SLAs, the best enforcement does not help: If the cloud provider overprovisions the underlying system, the user has to bear large costs, making the cloud provider less competitive and encouraging the user to take her business elsewhere. If the cloud provider underprovisions the underlying system, the cloud provider has to pay penalties for missed SLAs and thus loses money in the long term or must raise prices to compensate.

In this paper, we address the problem of *selecting* and *enforcing* SLAs for ad-hoc analytical queries over systems with multiple nodes. We develop SLAOrchestrator, a system that enables a cloud provider to offer query-level, performance SLAs for ad-hoc data analytics. Instead of relying on outside-generated SLAs [12, 11, 58], SLAOrchestrator uses our PSLAManager from prior work [43] to show the user what is possible and the price tag associated with various options. SLAOrchestrator generates, updates over time, and enforces SLAs in a way that successfully brings down the cost, close to that of the original service without SLAs.

Figure 1 shows our system in action given a set of random tenants and EC2 prices.[1] The x-axis shows time and the y-axis shows the ratio of the service cost with SLAs to the service cost without SLAs. When we add performance SLAs to Amazon EMR and let the cloud provision the number of Virtual Machines (VMs) purchased under the covers, costs grow dramatically either due to SLA violations (EMR+SLAs) or over-provisioning (EMR+SLAs+Buffer). Since guarantees depend on the quality of the SLAs (measured by how close runtime estimates are to the real runtimes on the purchased resources), a key component of our approach is *to improve SLAs over time* (EMR+Improving SLAs). We complement these improving SLAs with *novel resource scheduling and provisioning algorithms* that minimize costs due to over- or under-provisioning given a per-query SLA (SLAOrchestrator).

SLAOrchestrator achieves its goal through three key techniques that form the core contributions of this work. First, SLAOrchestrator is designed on the core idea of a double nested, learning loop. In the outer loop, every time a tenant arrives, the system generates a performance SLA given its current model of query execution times. That model improves over time as more tenants use the system. The SLA is in effect for the duration of a *query session*, which is the time from the moment a user purchases an SLA and issues their first query until the user stops their data analysis and leaves the system. In the inner loop, SLAOrchestrator continuously learns from user workloads to improve query scheduling and resource provisioning decisions and reduce costs during query sessions. To drive this inner loop, we introduce a new subsystem, that we call *PerfEnforce*. We present the overall system architecture in Section 2.

Second, the PerfEnforce subsystem comprises a new type of query scheduler. Unlike traditional schedulers, which must arbitrate resource access and manage contention, PerfEnforce's scheduler operates in the context of seemingly unbounded, elastic cloud resources. Its goal is *cost-effectiveness*. It schedules queries in a manner that minimizes over- and under-provisioning overheads. We develop and evaluate four variants of the scheduler. The first variant is based on a PI controller. Two variants model the problem as either a contextual or non-contextual multi-armed bandit (MAB) [52]. The last variant models the problem as an online learning problem. We present the query scheduler in Section 3.

[1] We present the detailed experimental setup in Section 5 and the exact SLA function in Section 3.1.

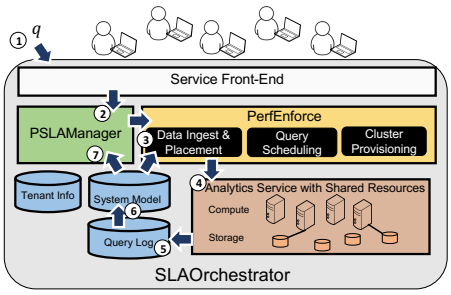


Figure 2: **SLAOrchestrator Architecture.**

Third, PerfEnforce also includes a new resource provisioning component. We evaluate two variants of resource provisioning: The first one strives to maintain a desired resource utilization level. The other one observes tenant query patterns and adjusts, accordingly, both the size of the overall resource pool and the tuning parameters of the query scheduler above. We present the resource provisioning algorithms in Section 4.

We evaluate all techniques in Section 5 and discuss related work in Section 6. As Figure 1 shows, SLAOrchestrator is able to reduce the costs associated with performance guarantees, bringing those costs down close to the basic service costs without guarantees.

2 System Architecture

Figure 2 shows SLAOrchestrator's system architecture. In this section, we present the details of that architecture and SLAOrchestrator's double nested learning loop.

2.1 System Components

SLAOrchestrator runs on top of a distributed, shared-nothing, data management and analytics engine (Analytics Service) such as Spark [7] or Hive [27]. We use our own Myria system [56] in the evaluation. Similar to how tenants use Amazon EMR today, in SLAOrchestrator, tenants upload their data to the service and analyze it by issuing declarative queries. While modern systems support complex queries, in this paper, we focus on relational select-project-join queries as proof-of-concept. However, there is nothing in our approach that precludes more complex queries in principal. On top of the Analytics Service, SLAOrchestrator includes an SLA generator (PSLAManager [43]), which generates performance SLAs for tenants. It also contains a dynamic scaling engine (PerfEnforce), which drives the scheduling and provisioning decisions for the underlying Analytics Service.

Analytics Service The back-end Analytics Service executes on a dynamically resizable pool of virtual

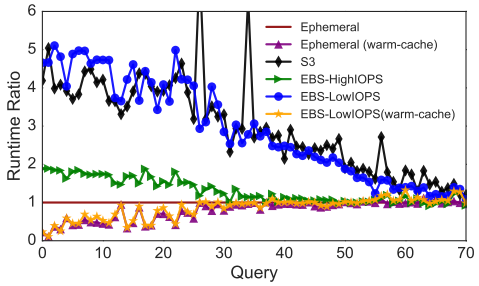


Figure 3: **Runtimes Compared to Local Storage.**

machines (VMs) running a resource manager such as YARN [6]. PerfEnforce uses that engine in a multi-tenant fashion and takes over all query scheduling decisions. When a tenant executes a query, PerfEnforce's query scheduling algorithm determines the number of containers needed to run the query. It then allocates that number of containers from the shared VM pool. Additionally, PerfEnforce's resource provisioning determines when to grow or shrink the pool.

Analytics Service: Tenant Isolation As is common in today's big data systems, each parallel partition of each query is a task that executes in a separate container. Each query submitted by each tenant thus gets allocated its own set of containers across the VMs. Furthermore, our design partitions each tenant's dataset and attaches individual data partitions to containers, allowing for a more isolated environment. In our experiments, we use YARN containers. We schedule one container per VM and thus use the terms interchangeably.

Analytics Service: Storage Once a user purchases an SLA and before they can query their data, PerfEnforce prepares their data by ingesting it into fast networked storage, EBS volumes in our prototype. Figure 3 motivates our choice. The figure shows the median query execution times across three runs for a variety of storage options available on Amazon Web Services (AWS). 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 queries shown are based on a 100SF TPC-H SSB dataset on Myria [56] running on 32 i2.xlarge instances. As the figure shows, fast networked storage, such as EBS-HighIOPS, provides performance competitive with ephemeral storage, even on a cold cache query, without the need to dynamically migrate (or replicate) data fragments as VMs are added and removed from the shared pool. This type of storage is also affordable at less than 20% of the cost of a VM. Because we seek dynamism and must support data-intensive processing, fast networked storage is appealing.

During query execution, PerfEnforce attaches EBS volumes to different VMs and detaches them as needed. Each EBS volume holds a partition of the data, resulting in a standard shared-nothing configuration. To avoid data shuffling overheads due to scaling, PerfEnforce ingests

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

Figure 4: **Example performance SLA provided by PSLAManager with one service tier. Additional service tiers would show similar query templates but with different prices and performance thresholds.**

multiple copies of each table. Each copy is partitioned across a subset of EBS volumes such that, when a query executes over a set of k containers, it uses the version of its data spread across k EBS volumes. Due to space constraints, we refer to our technical report for further details on EBS data placement and its negligible impact on performance [45].

SLA Generation To generate SLAs, we use a system from our prior work, the PSLAManager [43], but our system could work with others. PSLAManager takes as input a database schema and statistics associated with a database instance for a tenant (we use the term user and tenant interchangeably). It generates a performance-based SLA specific to a database instance as shown in Figure 4 for the TPC-H Star Schema Benchmark [42]. Each tier has a fixed hourly price, which maps to a pre-defined set of storage and compute resources, along with sets of grouped queries where each group contains a time threshold ("Runtime" in the figure). The time threshold represents the performance guarantee for its respective group of queries and corresponds to *query time estimates* made by the SLA generator for the corresponding resource configuration. For each resource configuration, we only consider varying the number of instances, but consistently use a standard network, and EBS-HighIOPS for storage across all configurations.

Each tier represents a performance summary for a specific set of containers the service can use for tenant queries, which we call a *configuration*. Tiers can correspond to different types and numbers of containers, but we use a single type in our experiments. We refer to all possible configurations that the system can use to execute a query as the set $configs$. For example, $config = \{2, 4, \ldots, 64\}$, represents all even numbers of containers up to a maximum of 64. The system shows tiers for a pre-defined subset of these configurations. Later, it can schedule queries using the full set of configurations. The price of each tier is at least the sum of the hourly cost of the containers and network storage.

When a tenant purchases a performance SLA, she unknowingly purchases a configuration. The system starts a *query session* for the tenant and the latter starts paying

the corresponding fixed hourly price. During the session, the tenant issues queries. The queries get queued up and execute one after the other, each one running in the entire set of containers in the purchased configuration. As we present in Section 3, PerfEnforce changes these allocations over time based on how fast they execute compared with the initial SLA time.

2.2 Double Nested Learning

To drive the SLA generation, SLAOrchestrator maintains a log of all past queries executed in the system. Initially, it executes queries from a 100GB dataset generated by the Parallel Data Generation Framework(PDGF) [47]. The system runs queries on all configurations that it will sell to populate the query log. With this information, SLAOrchestrator builds a model of query execution times. Each query is represented by a feature vector. Features correspond to query plan properties including the number of tables being joined, their sizes, the query cost estimates from the query optimizer, the number of containers in the configuration, etc. SLAOrchestrator learns a function from that feature vector to a query execution time. In our work, we use a simple linear model as in prior work [43, 55]. 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) [9] given the query features, x_q: $y(x_q, w) = \sum_{d=1}^{D} w_d \cdot x_{q_d}$.

With our previous PSLAManager work [43], we observed that when a new tenant joins the system, estimates for that tenant's queries are likely to be inaccurate because the system has limited information about the tenant data and queries (only statistics on base data). However, as the tenant starts to execute queries, the system can quickly learn the properties of the data and can specialize its model to that data. PerfEnforce uses this information to dynamically adjust query scheduling and resource provisioning decisions in the context of an existing SLA. We call this the **Inner Learning Loop**. The effect of this learning is also that the system updates the SLA that it offers after each query session. This is SLAOrchestrator's **Outer Learning Loop**. The benefit of more precise SLAs to tenants is the overall reduction in the service cost. We use TensorFlow [1] to build this model and train on the PDGF dataset. We generate 4000 queries (500 per configuration) and record the features as well as runtimes into the System Model.

Figure 2 shows in more detail how SLAOrchestrator components interact with one another. Steps 1 through 6 denote the **Inner Learning Loop**: (1) Each tenant query, q is issued through the service front-end. (2) PSLAManager determines q's promised SLA time based on the service tier that the user previously purchased. (3) PerfEnforce uses query scheduling algorithms in conjunction with the System Model to determine the number of containers to schedule for q. (4) PerfEnforce schedules q on the Analytics Service. (5) The Analytics Service sends metadata about the query to the Query Log. (6) The System Model parses the Query Log metadata and stores features for the learning models. Once a tenant completes their session, SLAOrchestrator initiates the **Outer Learning Loop**. In Step 7, the PSLAManager system takes the information from the System Model and generates an improved SLA.

In the next two sections, we focus on the PerfEnforce subsystem and its query scheduling (Section 3) and resource provisioning (Section 4) algorithms, which are part of SLAOrchestrator's inner learning loop.

3 Dynamic Query Scheduling

Every time a new tenant purchases a service tier, PerfEnforce begins a query session for that tenant. The initial state of the query session indicates the configuration (i.e., number of containers) that corresponds to the purchased service tier. Many sessions are active at the same time and PerfEnforce receives streams of queries from these active tenants. Each query is associated with a possibly imperfect SLA. That is, the query may run significantly faster or slower than the SLA time if scheduled on the purchased set of containers. PerfEnforce's goal is to determine how many containers to actually use for each query with the goal to minimize operation costs. In this section, we present PerfEnforce's query scheduling algorithm.

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 :

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

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

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

where $cost_t(resources)$ represents the cost of 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. In this paper, we use the following formulation:

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

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,

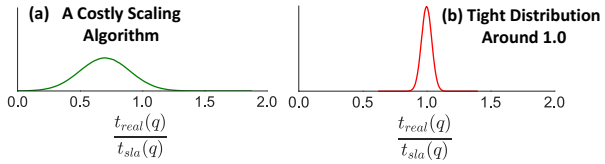


Figure 5: **Examples of Distributions of Query Performance Ratios**

$t_{sla}(q)$ is the SLA time of q, p_u is the 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 container resource costs. $\mathcal{S}$ is a step function that rounds up and truncates values. This step function is inspired by real SLAs in cloud services that incur penalties based on availability outages [8, 51, 53].

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 number of containers from the shared pool to allocate to the query. PerfEnforce begins with using the number of containers that corresponds to the purchased service tier. It observes the resulting query runtimes and dynamically adjusts the number of containers for subsequent queries by using a *scaling* algorithm. It runs a scaling algorithm separately for each tenant.

To minimize resource costs, the scaling algorithm should schedule queries on the smallest possible number of containers. To avoid SLA penalties, however, it must schedule queries on sufficiently large numbers of containers 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 5. 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 5a) or causing a large number of SLA violations. A good query scheduling algorithm should yield a tight distribution as in Figure 5b).

3.2.1 Reactive Scaling Algorithms

A reactive algorithm observes errors between the real and SLA runtimes and adjusts the number of containers accordingly for each subsequent query. 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 [32, 34, 36, 39].

A limitation of the these techniques is that the configuration size chosen for a new query depends only on the rewards or errors of previous queries ignoring the features of the current query. We use the reactive methods as baseline.

Proportional Integral Control (PI). Feedback control [29] 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 containers to use for 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, $y(t)$, 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 containers 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 containers 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: $u(t+1) = u(0) + \sum_{x=0}^{t} k_i e(x) + k_p e(t)$

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 [52]. In our setting, each arm is a configuration from the set $configs$. The arm choice is the decision to schedule the next query using 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 [9]. At each timestep, the system must thus choose to either select the arm with the highest estimated reward (*exploitation*) or try another arm (*exploration*) in order to acquire more information and maximizing the reward across all timesteps [52].

To help balance between exploration and exploitation, we use a heuristic known as *Thompson Sampling* [10]. 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, $arg\,max_a \mathbb{E}[P(r_t|\hat{\theta}_\alpha)]$.

3.2.2 Proactive Scaling Algorithms

To address the limitations of the reactive techniques, 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 costs, estimated rows, estimated width, and the number of workers scheduled to run the query).

Contextual Multi-Arm Bandit (CMAB). This approach is a variant of the multi-armed bandit problem that includes contextual information. 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 [10]. 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 [23] as used in BanditDB [39]. 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. PerfEnforce runs the "warm-up" session at the start of the query session, which could impact performance for some queries.

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. We build a single model of query execution times with the configuration size as a feature. 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.

As described in the previous section, SLAOrchestrator maintains a model of query execution time that it uses for SLA generation. The idea here is for PerfEnforce to *continuously update* that model, during a tenant's query session, based on the measured query execution times. 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: $w^{(\tau+1)} = w^{(\tau)} - \eta\nabla E$ [9]. Where τ represents the nth data point and η represents the learning rate. Importantly, PerfEnforce maintains a separate model of query execution time for each dataset so as to specialize its model to the properties of that dataset. If the underlying data significantly changes, the model could take time to adjust to changes, depending on the learning rate. Since we primarily focus on analytic sessions, we do not evaluate how this model adapts to updates. Training this model is relatively cheap, taking approximately 2.38s for a single epoch. Each prediction takes $\sim$10ms.

Setpoint Adjustment With all algorithms above, PerfEnforce strives to schedule queries such that their performance ratios form a tight distribution around a desired setpoint. An important question is how to tune the value of that setpoint. If the setpoint is 1.0 and the mean of the distribution falls on that setpoint, 50% of all queries will miss their SLA times. The setpoint can be lowered such that more, perhaps 90% of all queries, meet their SLA time. Lowering the setpoint, however, will increase the number of containers used for those queries and will thus raise resource costs. In SLAOrchestrator, we adjust the setpoint dynamically. We do so at the same time as we make cluster provisioning decisions as described next.

4 Dynamic Provisioning

With the above query scheduler, the total number of containers needed to service the active set of tenants varies over time. To reduce operation costs, PerfEnforce includes a resource provisioning component that determines when to grow and shrink the *shared* pool of compute resources. Provisioning is particularly challenging

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: $\quad AvgUtilization \leftarrow \frac{1}{|V|} \sum_{v \in V} Utilization(v)$
6: $\quad e_p = \frac{AvgUtilization - Z}{Z} \cdot |V|$
7: $\quad V_{new} = V_{init} + (e_i * V_{k_i}) + (e_p * V_{k_p})$
8: $\quad$ **if** $V_{new} - |V| > 0$ **then**
9: $\quad\quad numAdd = V_{new} - |V|$
10: $\quad\quad$ **for** $a \rightarrow 1, numAdd$ **do**
11: $\quad\quad\quad V = V \cup AddVM()$
12: $\quad$ **else if** $V_{new} - |V| < 0$ **then**
13: $\quad\quad R = 0$
14: $\quad\quad numRemove = |V| - V_{new}$
15: $\quad\quad$ **for** each $v \in V$ **do**
16: $\quad\quad\quad$ **if** $isFree(v)$ and $R < numRemove$ **then**
17: $\quad\quad\quad\quad RemoveVM(v)$
18: $\quad\quad\quad\quad V = V - \{v\}$
19: $\quad\quad\quad\quad R+ = 1$
20: $\quad e_i+ = e_p$

Algorithm 2 Provisioning with User Workload Simulations

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

as it can take time to spin up new virtual machines. We observe that it takes approximately 12 seconds to spin up a virtual machine with a pre-loaded image on Amazon. We consider this start up time throughout our evaluation.

Utilization 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 [3, 8, 22, 50].

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 imply a higher demand for resources [14].

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. Besides wall clock time, 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. 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.

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. 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 has been recently used in the "What-If" engine from Tempo [54], where the goal is to simulate the performance of many configurations of the MapReduce Resource Manager. We aim to understand how such a provisioning algorithm in combination with a learning query scheduler can help make profitable decisions in a multi-tenant service.

In this 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 gener-

ates 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. During these simulations, we also evaluate the costs of dynamically shifting the setpoint. In general, these simulations help PerfEnforce discover whether setpoint adjustments are necessary for active tenants or whether nodes should be added or 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 and setpoints, $sharedConfigs$. For a $sharedConfig$, 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

We run SLAOrchestrator and execute all queries on Amazon EC2 using i2.xlarge (4 ECU, 30 GB Memory) instance types priced at $0.12/hr. We consider eight types of query scheduling configurations, each with a different number of compute instances: `configs` $= \{4, 8, 12, 16, 20, 24, 28, 32\}$. For multi-tenant experiments, we run simulations with up to thousands of servers and use the query times measured on EC2. For our underlying shared-nothing, database management system, we use Myria [56]. 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. Unless stated otherwise, all the query workloads throughout the evaluation are generated in this fashion.

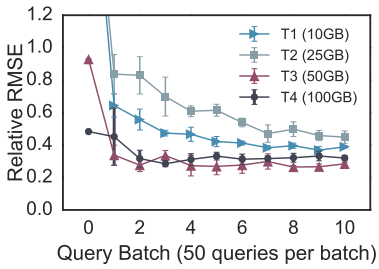


Figure 6: **SLA improvements across query sessions**

5.1 Evaluation of SLA Predictions

A key tenet of SLAOrchestrator is the idea that the system should update SLAs because they rapidly improve as a tenant queries a database. We validate this hypothesis in this section. Figure 6 shows the error of the SLA predictions for four tenants, each with a different, random star schema [47] and database instance: T1 = 10GB, T2 = 25GB, T3 = 50GB, T4= 100GB. We generate a set of SPJ queries with random selection predicates for each tenant. As tenants execute queries, SLAOrchestrator updates the query time model separately for each database. After each query batch, PSLAManager re-generates an updated SLA. As the figure shows, in all cases, the RRMSEs (relative root mean squared errors) between the real runtimes and the predicted SLA runtimes decreases rapidly after the first batch and then improves more slowly. We compute the error on a sample of queries generated by the PSLAManager for the tenant SLA. 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 generated 100GB PDGF dataset. We observe a higher initial RRMSE error (approx. 2.4-2.5) for tenants T1 and T2 with the smaller databases.

Figure 7 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

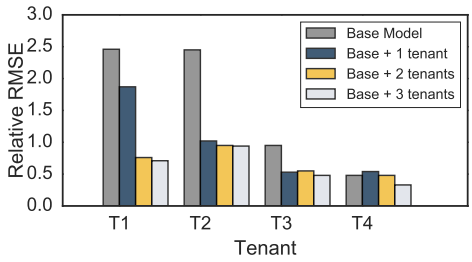


Figure 7: **SLA improvements across tenant sessions**

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 (later in Section 5.4, we consider dynamic setpoint tuning). 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 TPC-H SSB benchmark [42]. We evaluate the algorithms using different-quality SLAs as shown in Table 1, which could correspond, for example, to different model qualities as shown in Section 5.1.

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 8. The y-axis represents the distance between μ and 1.0, while the x-axis displays the standard deviation of the query performance distributions. 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. If any technique's parameters result in a distribution with a higher σ or a μ farther from 1.0, this error impacts cost, which ultimately depends on the cost function. 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. There are no best set of parameters that work across all workloads.

In the second row of Figure 8, we show the average distributions for MAB, CMAB, and online learning across the same set of SLA types and tenant query sessions. 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.

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 con-

SLA	Description
Small Gaussian Error (SG SLA)	SLA assumes a good prediction model and tests sensitivity to small errors (or variance) in query times. Generated by taking the real query execution times at the purchased tier and adding a small Gaussian error: $\mathcal{N} = (\sigma = 0.1 * t_{real}(q), \mu = 0)$.
Positive/Negative Gaussian Errors for Large Joins (PLJ/NLJ SLA)	We skew SLA runtimes for some query types. We introduce large positive/negative errors to the real runtimes on queries with a large number of joins (>3 joins) and with a runtime >100 seconds. We update the runtime to $t_{real}(q) + \|e\|$ (or $-\|e\|$), 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 SG SLA.
Initial SLA	This is the least accurate SLA, where runtimes are generated by an initial offline-trained model.

Table 1: **SLAs used in experiments.**

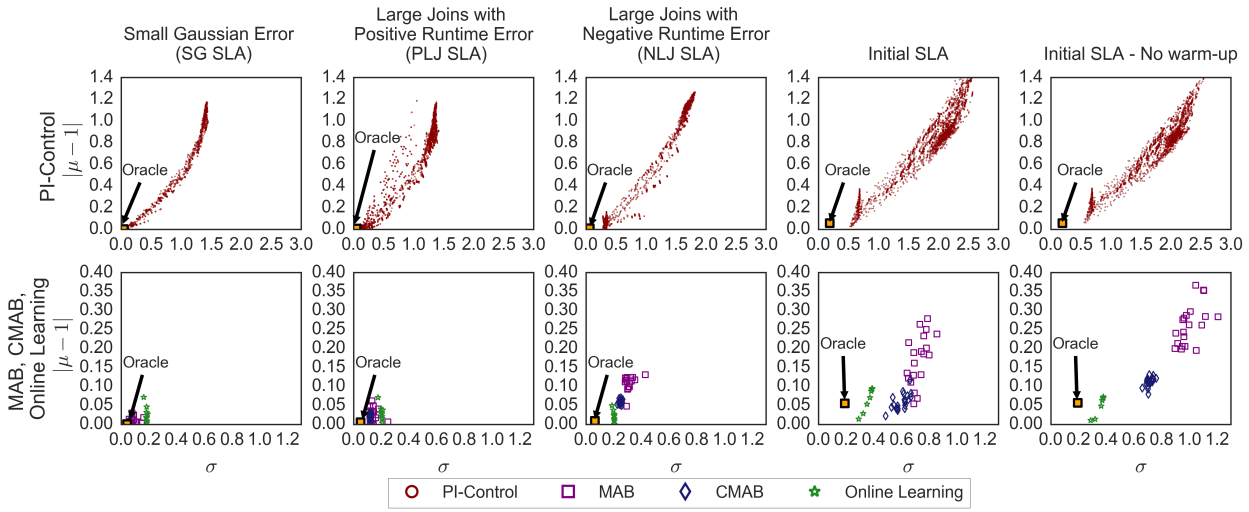


Figure 8: **Evaluation of PI-Controller, MAB, CMAB and online learning scheduling algorithms**

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

Table 2: **Ratio distributions during slow down**

troller is ill-suited to our problem and we do not consider it further.

We now evaluate how query scheduling algorithms can adapt to changing conditions. Recall, the goal of these query schedulers is to ensure a query performance ratio distribution close to 1.0. We 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. Table 2 shows that the online learning technique reacts the fastest to this change in conditions, leading to an overall mean performance ratio closest to 1.0. We omit additional experiments with different changing workloads due to space constraints.

5.3 Evaluation of Provisioning Algorithms

We first evaluate each provisioning algorithm in combination with the Oracle query scheduler to ensure that

Notation	Description
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

Table 3: **Parameters of multi-tenant experiments**

query runtime penalties are not a side-effect of the query scheduler's mispredictions. We launch each multi-tenant tenant session based on session parameters summarized in Table 3.

We introduce up to 100 tenants in a session and simulate a shared cluster with thousands of containers/VMs. 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 VMs from the pool. Our step function $\mathcal{S}$ provides no service credit if the system misses the runtime by 10%. For each additional 20% increment and given a

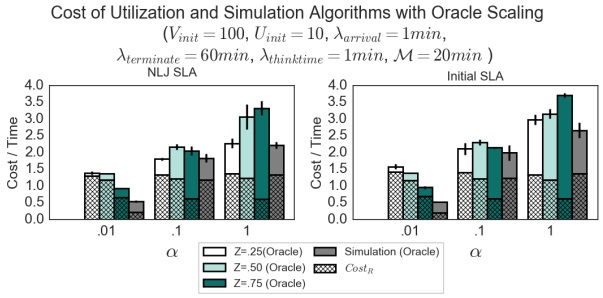


Figure 9: **Comparing utilization-based and simulation-based provisioning in conjunction with an Oracle query scheduler. The hash pattern represents the proportion of the cost due to $Cost_R$**

threshold from x% to y%, we increase the credit to y%.

As described in Section 2, each submitted query gets allocated a set of containers. We schedule one container (running a Myria process) per VM. For each query, the system assigns the tenant's EBS volumes to a set of VMs in the pool. After the query completes, the volumes are detached from the VMs, making them available to other tenants. We find it takes 4 seconds to mount a volume to a VM. Detaching takes approximately 11 seconds. We include these delays in the experiments.

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 VMs with 10 initial tenants and set the provisioning monitoring time to $20min$. Figure 9 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 SLA penalty compared to the 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. Error bars show variance across 10 runs. As expected, the utilization-based method requires tuning depending on the α value. Simulation-based provisioning has the double-benefit of avoiding any tuning and more cost effectively provisioning shared resources compared to the utilization-based approach.

Overhead of Unpredictable Workloads 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 10 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.

Combining Scheduling and Provisioning We now evaluate the performance of simulation-based provisioning in conjunction with various query scheduling algorithms on the initial SLA. In Figure 11, 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 pool of 320 VMs, allowing enough room to have each initial tenant schedule queries on up to 32 containers. 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, 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.

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 run-

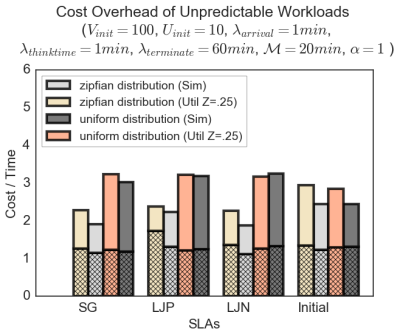


Figure 10: **Measuring the overhead of unpredictable workloads**

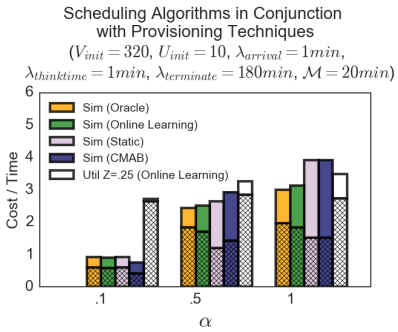


Figure 11: **Comparing the costs of query scheduling in conjunction with provisioning**

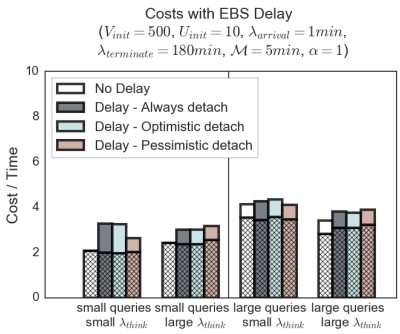
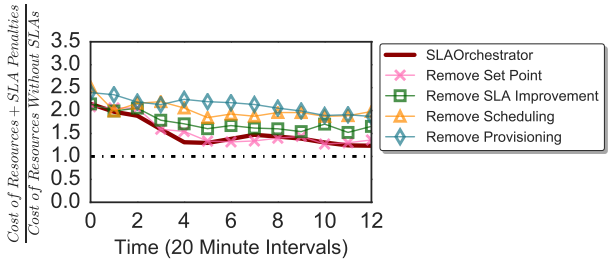


Figure 12: **Overhead in costs due to EBS delays and benefits of pessimistic detach heuristic.**

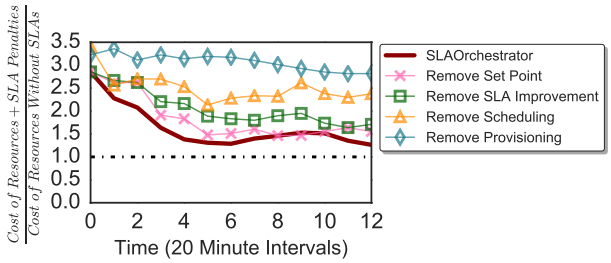
times are high, the time to attach/detach is a smaller overhead. Additionally, $\lambda_{thinktime}$ is also a factor to consider. If the think times are long, it provides sufficient amount of time for EBS volumes to successfully detach from VMs once a query has completed. Short think times that are 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 12, 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.



(a) $\alpha = 2$



(b) $\alpha = 3$

Figure 13: **Performance when disabling different SLAOrchestrator optimizations**

5.4 Dynamic Setpoint Tuning

Finally, we evaluate the benefits of dynamic setpoints together with the relative benefits of the other optimizations. Figure 13a shows the results. In the figure, we start with SLAOrchestrator as initially shown in Figure 1. We then remove optimizations one at a time in order.

First, we remove the ability to use dynamic setpoints, followed by removing SLA improvements, scheduling and provisioning. We remove these optimizations to run SLAOrchestrator as a simpler multi-tenant system. To emphasize the differences between optimizations, we use $\alpha = 2$ and $\alpha = 3$. In this experiment, we start the session with 5 tenants and 80 VMs (ensuring 16 nodes per tenant, the amount they have purchased). New tenants arrive approximately every 5 minutes, and tenants finish their session after 180 minutes. As seen in the figures, removing each optimization increases the cost. This is especially apparent for $\alpha = 3$, where SLAOrchestrator decreases the cost by up to 59% compared to the case with no optimizations.

5.5 Evaluation Discussion

The SPJ query workloads we use throughout the evaluation allow us to demonstrate a proof of concept for SLAOrchestrator. Incorporating more complex query workloads (e.g., considering aggregates and subqueries), would impact the online learning and CMAB techniques as they would require more advanced models and more extensive feature engineering than those considered in this work. Second, for a more thorough provisioning evaluation, running SLAOrchestrator against real tenant traces would help to better understand how the system would behave under more bursty workloads.

6 Related Work

Elastic Scaling for Performance Guarantees Performance guarantees have traditionally been the focus of real-time database systems [31], 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 [12, 11, 58], possibly postponing or even simply rejecting queries. Work by Das et. al [14], 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 [55], CherryPick [2] and CloudScale's [50] goal is to also find good configurations for analytical workloads, but they require representative workloads or depend on repeating tenant usage patterns. Morpheus [30] 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 [34], and systems such as TIRAMOLA [32] and BanditDB [39] which leverage reinforcement learning techniques to make optimal decisions. In addition, many other existing solutions leverage decisions based on meeting resource utilization goal [13, 16, 20, 41, 53, 59]. The Tempo [54] 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* [18, 37, 35]. Related work addresses this by mitigating bad tenant packing by either finding a good initial tenant placement strategy or dynamically migrating tenants [16, 18, 33, 35, 53, 57]. 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 [16], a system based on Paragon [15], 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 [58], building gray-box performance models [21], using historical traces of previous workloads [19], using a broad set of benchmarks to profile resources [60], or running smaller samples of the workload with a low overhead [55]. 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. [28] 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. [39], which allows machines to learn an optimal time to wait before they shut down. Neural networks for dynamic allocation [40] or dynamic provisioning [46] 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

We presented SLAOrchestrator, a new system designed to minimize the price of performance SLAs in cloud an-

alytics systems. SLAOrchestrator uses a double learning loop that improves SLAs and resource management over time. To support the latter, the system also includes an efficient combination of elastic query scheduling and multi-tenant resource provisioning algorithms that work toward minimizing service costs. Experiments demonstrate that SLAOrchestrator dramatically reduces service costs for a common type of per-query latency SLAs.

Acknowledgments: This project was supported in part by NSF grants IIS-1247469 and IIS-1524535, and the Intel Science and Technology Center for Big Data.

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 RDS. http://aws.amazon.com/rds/.

[4] Amazon AWS. http://aws.amazon.com/.

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

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

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

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

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

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

[11] 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.

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

[13] 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.

[14] 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.

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

[16] 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.

[17] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proc. of the SIGMOD Conf.*, pages 301–312, 2011.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] B. He, M. Yang, Z. Guo, R. Chen, W. Lin, B. Su, H. Wang, and L. Zhou. Wave computing in the cloud. In *Proceedings of HotOS'09: 12th Workshop on Hot Topics in Operating Systems, May 18-20, 2009, Monte Verità, Switzerland*, 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] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 261–272, 2011.

[27] Hive. http://hadoop.apache.org/hive/.

[28] 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.

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

[30] 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.

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

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

[33] 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.

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

[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. Wisedb: A learning-based workload management advisor for cloud databases. *Proc. VLDB Endow.*, 9(10), June 2016.

[39] 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.

[40] 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.

[41] 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.

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

[43] 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.

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

[45] Perfenforce: A dynamic scaling engine for analytics with performance guarantees. http://myria.cs.washington.edu/publications/perfenforce_tech_report_2018.pdf.

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

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

[48] 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.

[49] J. Rogers, O. Papaemmanouil, and U. Cetintemel. A generic auto-provisioning framework for cloud databases. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 63–68, 2010.

[50] 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.

[51] Sla for azure cosmos db. https://azure.microsoft.com/en-us/support/legal/sla/cosmos-db/v1_0/.

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

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.