Releasing Cloud Databases from the Chains of Performance Prediction Models

Ryan Marcus
Brandeis University
ryan@cs.brandeis.edu

Olga Papaemmanouil
Brandeis University
olga@cs.brandeis.edu

ABSTRACT

The onset of cloud computing has brought about computing power that can be provisioned and released on-demand. This capability has drastically increased the complexity of workload and resource management for database applications. Existing solutions rely on query latency prediction models, which are notoriously inaccurate in cloud environments. We argue for a substantial shift away from query performance prediction models and towards machine learning techniques that directly model the monetary cost of using cloud resources and processing query workloads on them. Towards this end, we sketch the design of a learning-based service for IaaS-deployed data management applications that uses reinforcement learning to learn, over time, low-cost policies for provisioning virtual machines and dispatching queries across them. Our service can effectively handle dynamic workloads and changes in resource availability, leading to applications that are continuously adaptable, cost effective, and performance aware. In this paper, we discuss several challenges involved in building such a service, and we present results from a proof-of-concept implementation of our approach.

1. INTRODUCTION

Infrastructure-as-a-Service (IaaS) providers offer low cost and on-demand computing and storage resources, allowing application to dynamically provision resources, i.e., procure and release them depending on the requirements of incoming workloads. Compared with traditional datacenters, this new approach allows applications to avoid static over or under provisioned systems by scaling up or down for spikes or decreases in demand. This is realized by the "pay as you go" model of IaaS cloud, in which applications pay only for the resources they use and only for as long as they use them.

However, taking advantage of these benefits remains a complex task for data management applications, as deploying and scaling an application on an IaaS cloud requires making a myriad of resource and workload decisions. Application developers must choose how many machines to provision, which queries to route to which machines, and how to schedule queries within machines. Minimizing and even predicting the cost of each of these decisions is a complex task, as the resource availability of each machine and the execution order of the queries within them have great impact on the execution time of query workloads. This complexity increases significantly if applications wish to meet certain performance goals (Service-Level-Objectives/SLOs).

Most IaaS providers assume their users will manually instigate a scaling action when their application becomes popular or during periods of decreased demand, and that they will deploy their own custom strategies for dispatching workloads to their reserved machines. Therefore, in many real-world applications, scaling and workload distributions decisions are still made based on rules-of-thumb, gut instinct, or, in the best cases, past data. Even when application developers grasp the complexity of cloud offerings, it is often still difficult to translate an application's performance goal (e.g., queries must complete within 5 minutes, or the average latency must be less than 10 minutes) into a cost effective resource configuration and workload distribution solution.

While this problem has been partially addressed in the literature, the landscape of solutions is fractured and incomplete. Many workload and resource management solutions are not *end-to-end*: they address only one issue, such as query routing to a reserved machine (e.g., [34]), scheduling within a single machine (e.g., [16]), or provisioning machines (e.g., [43]), without addressing the others. However, applications must address *all* of these challenges, and integrating multiple solutions is extremely difficult due to different assumptions made by each individual technique.

More importantly, even solutions that span several of the decisions that must be made by cloud applications depend on a query latency prediction model (e.g., [10, 16, 17, 22, 24, 29, 31, 32, 36, 37, 47, 51, 52]). This dependency is problematic for two reasons. First, many latency prediction models (e.g., [8, 20, 50]) depend on seeing each "query template" beforehand in a training phase, leading to poor predictions on previously-unseen queries. Second, *accurate* query latency prediction is very challenging. State-of-the-art results for predicting the performance of concurrent queries executed on a single node achieve 85% accuracy for known query types (e.g., [20]) and 75% accuracy for previously unseen queries (e.g., [19]). A cloud setting only brings about additional complications like "noisy neighbors" (e.g., [11, 39]) and requires training these models on virtual machines with vastly different underlying resource configurations.

8th Biennial Conference on Innovative Data Systems Research (CIDR '17) January 8-11, 2017 , Chaminade, California, USA.

In this paper, we argue that both the status quo solution of scaling based on rules-of-thumb, human-triggered events, or methods that rely on a query performance prediction models, fail to fully achieve the promise of IaaS-deployed cloud databases. Humans may drastically mispredict the best times to scale and what scale to achieve. Latency prediction based techniques suffer from a large range of accuracy problems that worsen with scale and unknown query types, inherently undermining the main objective: estimating the *cost* of using cloud resources while meeting performance goals. Hence, instead of explicitly modeling the latency of each query and then using that latency to estimate the cost of various scheduling or provisioning decisions, we propose *modeling the* cost *of these actions directly.*

We envision a new class of services for IaaS-deployed data management applications that:

- Accept application-defined performance goals and tune themselves for these goals.
- Adapt and continuously learn from shifts in query workloads, constantly aiming for low-cost deployments.
- Automatically scale resources and distribute incoming query workloads.
- Refuse to *explicitly* model query latency, which is impossibly problematic in a cloud setting, and instead build models of the cost of various actions, which will *implicitly* capture query latency information.
- Balance *exploration and exploitation*, automatically trying out new resource configurations while taking advantage of prior knowledge.

In this paper, we discuss the complexities of implementing our vision, and we give an imperfect but illustrative proof-of-concept workload and resource management and provisioning service. Our system, called *Bandit*, is decoupled from query performance prediction models. Instead, it utilizes reinforcement learning algorithms to learn on the fly (and improve over time) *cost-effective* performance management policies that are aware of application-defined service-level objectives (SLOs).

Bandit learns models that capture the relationship between workload management decisions and their monetary cost. These models relieve developers from the tedious tasks of system scaling, query routing, and scheduling: Bandit automatically scales up and down the pool of reserved machines and decides the processing sites of incoming queries without *any* prior knowledge of the incoming workloads (e.g., templates, tables, latency estimates). *Bandit demonstrates how machine learning techniques can produce systems that naturally adapt to changes in query arrival rates and dynamic resource configurations, while handling diverse application-defined performance goals, all without relying on any performance prediction model.*

The rest of the paper is organized as follows. Section 2 describes the high-level system model of Bandit. Section 3 highlights the parallels between problems studied in reinforcement learning and the problems faced by the cloud database research community, and describes how Bandit uses reinforcement learning to address resource provisioning and workload management challenges. Section 4 showcases preliminary results from our proof-of-concept implementation. We discuss related works in Section 5. Finally, we conclude in Section 6.

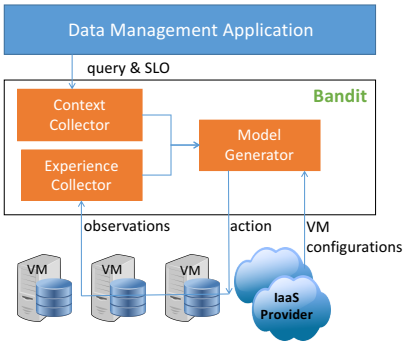


Figure 1: The Bandit system model

2. SYSTEM MODEL

We envision our service lying between the IaaS provider and a data management application as shown in Figure 1. We assume applications are deployed on an IaaS cloud (i.e., AWS [2], Azure [4]) and hence they have full control of provisioning virtual machines (VMs) and distributing incoming queries across them.[1]

We serve OLAP (analytic) workloads with read-only queries. Each reserved VM runs on either full replicas of the database or partitioned tables (where partition could also be replicated). We also assume that the application aims to meet a Service-Level-Objective (SLO) promised to its end-users, and if the SLO is not met, a penalty function defines the monetary cost of the violation.

Our system facilitates online scheduling on behalf of the application: queries arrive one at a time with an *unknown* arrival rate, and our service will schedule their execution either on one of the existing VMs or a newly provisioned VM. Queries could be instances of query templates (e.g., TPC-H), but these templates are *unknown* a-priori. Bandit seeks to minimize the monetary cost paid by the application, which includes the cost for renting the reserved VMs as well as any SLO violation fees. VMs cost a fix dollar amount for a given rent period, and VMs of different types (i.e., different resource configurations) are offered at different costs.

The application interacts with Bandit by defining an SLO, as well as a penalty function that specifies the monetary cost of failing to achieve the SLO. Bandit supports application-provided SLOs on the query *and* workload level. Examples include (a) a deadline for each incoming query, (b) an upper bound on the maximum or average latency of the queries submitted so far, or (c) a deadline on a percentile of the queries within a specific time period (e.g., 99% of submitted queries within each hour must complete within five minutes). If the SLO is defined over a set of queries, Bandit aims to minimize the *cumulative* cost of executing this query set. *Bandit is agnostic to the performance metric of the SLO, and requires only a penalty function mapping the query latencies to penalties.*

[1]Database-as-a-Service (DaaS) products [1, 3, 4] available today adopt a different model where these tasks are administered by the cloud provider and hence are outside the scope of this work.

During runtime, the application forwards each incoming query to Bandit, which executes the query on a VM and returns the results. In the back-end, Bandit interacts with the underlying IaaS to provision VMs and execute queries. Specifically, it leverages a context-aware reinforcement learning approach that uses features of the query as well as information about the underlying VMs to decide which machine should process each new query, or if machines should be provisioned or released. We collectively refer to these features as the *context* of the decision (collected by the *Context Collector* in Figure 1). Bandit records the cost of each past decision and the context that decision was made in into a set of *observations*. This is implemented by the *Experience Collector* module in Figure 1. By continuously collecting and using past observations, Bandit improves its decisions and converges to a model that balances the number and types of machines provisioned against any penalty fees in order to make low-cost performance management decisions for each incoming query.

3. REINFORCEMENT LEARNING

Generally speaking, reinforcement learning problems are ones in which an *agent* exists in a *state*, and selects from a number of *actions*. Based on the state and action selected, the agent receives a *reward* and is placed into a new state. The agent's goal is to use information about its current state and its past experience to maximize reward over time.

It is not difficult to draw parallels between these concepts and the challenges faced by users in cloud environments. In the cloud database context, the agent is the application, the state is the currently provisioned set of machines and the queries they are processing, the actions are a set of provisioning and query dispatching decisions, and the reward is inversely proportional to the cost paid to the IaaS provider. Next, we formalize this mapping and show how techniques from the reinforcement learning literature can be applied.

3.1 Contextual Multi-Armed Bandits

One abstraction developed from the field of reinforcement learning is the contextual multi-armed bandit (CMAB) [15]. Here, a gambler (agent) plays on a row of slot machines (one-armed bandits) and must decide which machines to play (i.e., which arms to pull) in order to maximize the sum of rewards earned through the sequence of arm pulls. In each round, the gambler decides which machine to play (action a) and observes the reward c of that action. The decision is made by observing a feature vector x (a.k.a. *context*) which summarizes information about the state of the machines at this iteration. The gambler then improves their strategy through the new observation $\{a, x, c\}$, which is added to the experience set $\mathcal{D}$. The gambler aims to collect information about how the feature vectors and rewards relate to each other so that they can predict the best machine to play next by looking at the feature vector.

CMABs in Bandit We model the workload and resource management problem as a *tiered network of CMABs*, illustrated in Figure 2. Each running VM corresponds to a slot machine (aka CMAB) in one of several tiers, where each tier represents a distinct VM configuration available through the IaaS provider. Tiers can be ordered based on price or performance/resource criteria. Each VM has three arms/actions: `Accept`, `Pass`, and `Down`. When a query enters the system, Bandit collects query-related features (the context) and asks

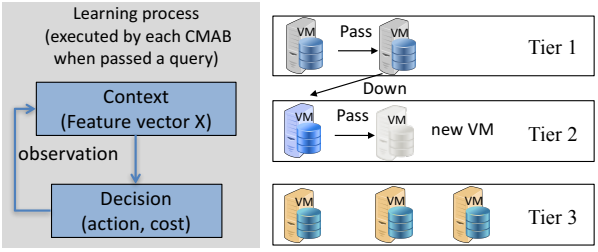


Figure 2: Bandit framework and an example decision process

the root CMAB (top left) to pick an action. The algorithm makes a decision based on the observed context and experience collected from past decisions. If the `Accept` action is selected, the query is added to that VM's execution queue. If the `Pass` action is selected, the query is passed to the next CMAB in the same tier. If there is no other CMAB on that tier, a new VM is provisioned and a corresponding CMAB is created. If the `Down` action is selected, the query is passed downwards to the first CMAB in the next tier. The last tier contains no `Down` arms. The network contains no cycles, and empty CMABs cannot select `Pass` (but may select `Down`), so a query will eventually be accepted by some CMAB. Note that the CMAB network can reside entirely inside of a single server, and queries do not need to be passed through a computer network.

After the query completes, the cost for each decision is determined. This includes (a) VM startup fees (if a new VM was provisioned) (b) the fees for processing that query on the VM and (c) any costs incurred from violating the SLO. Formally,

$$c = f_s + f_r \times l_q + p(q)$$

where f_s is the VM startup fees, f_r is the rent rate for the VM that executed the query, l_q is the query's execution time, and $p(q)$ calculates applicable penalties. Note that, after the query has completed, the query latency l_q is known.

We use the final cost c as a measure of how good the decisions made by the CMAB were: lower costs means better decisions. Each completed query and its associated cost c, along with the action selected by each CMAB a and the context x of the CMAB at the time the decision was made, can be used to "backpropogate" new information to all the CMABs involved in processing the query.

Specifically, when a query completes, each CMAB that the query passed through records (1) its context x when the query arrived, (2) the action selected a, and (3) the cost incurred by the network as a whole to execute the query c, forming a new observation $\{a, x, c\}$. Each CMAB adds this new observation to its set of experiences $\mathcal{D}$, thus providing each CMAB with additional information. If the cost of a CMAB taking action a in context x produced a particularly high cost, the CMAB will be less likely to select that same action in similar contexts. If the cost was particularly low, the CMAB will be more likely to select that action in similar situations. We explain the details of action selection in Section 3.3. As more queries move through the system, each CMAB's experience grows larger, and the system as a whole learns to make more cost-effective decisions.

Example To illustrate this process, imagine a CMAB network with limited prior experience. So far, the network has

only received queries that are too computationally expensive to be efficiently processed on the first tier of VMs, but the network has chosen to execute every query on one of two VMs in the first tier. As a result, each CMAB has observed a high cost for each query, since each query failed to meet its SLO. Now, when a new query arrives, the CMABs on the first tier are less likely to select the `Accept` option because their experience tells them it is associated with high cost. Eventually, the CMAB will select the `Down` action. When it does so, the query will be accepted on a VM in the second tier, and the original VM will associate a lower cost with its context and the `Down` action, making `Down` more likely to be selected in the future. In this way, the system *learns* that certain queries are too expensive to be processed on the cheaper tier of VMs.

Cost Propagation A tiered network of CMABs where costs are "backpropogated" to all involved VMs can automatically learn to handle many complexities found in cloud environments. Since each CMAB involved in placing a query receives the same cost, the entire network can learn advanced strategies. One example of a complexity "automatically" handled by the tiered network of CMABs is passing queries to machines with an appropriately warm cache. If the first machine in the network has information cached that is helpful in processing queries of type A, and the second machine in the network has information cached that is helpful for processing queries of type B, then the first machine will receive a low cost from the `Accept` arm when processing a query of type A, and the first machine will receive a low cost from the `Pass` arm when processing queries of type B. Since the costs are shared, searching for a low-cost strategy at each CMAB individually is equivalent to searching for a low-cost strategy for the network as a whole.

Query scheduling With only `Accept`, `Down`, and `Pass` arms, a VM would never be able to place a new query ahead of a query that had already been accepted. Hence, the system is restricted to using a FIFO queue at each machine. To address this limitation and allow for query reordering, one can "split" the `Accept` arm into smaller arms representing various *priorities*, e.g. `Accept High`, `Accept Medium`, and `Accept Low`. Each of the new accept arms represent placing a query into a high, medium, or low priority queue respectively. When a processing query completes, the head of the high priority queue is processed next. If the high queue is empty, then the head of the medium priority queue is processed, etc. While this modification allows Bandit to reorder incoming queries (albeit to a limited extent), it drastically increases the complexity of the problem by creating many more options to be explored by our learning system.

3.2 Context Features

In order to take advantage of the CMAB abstraction, and most other reinforcement learning models, we must identify *features* that can be extracted upon arrival of a query. These features serve as a proxy for the current state of the system, so they must contain enough information for an intelligent agent to learn the relationship between these features, actions, and monetary cost. In the CMAB abstraction, these features will compose the context x.

Our context includes a set of query and VM related features. It is critical to remember that the goal is to model the *monetary cost* of an action, not the exact latency of a particular query. We can thus expand our field of view beyond metrics that are direct casual factors of query latency. While none of our selected features would be enough *on their own* to indicate the cost of an action, and while some features may seem only tangentially related to the cost of the action, their *combination* creates a description of the context that is sufficient to *model* the cost of each action. This view allows us to work with features that may seem to only be *correlated with*, as opposed to being a *direct cause of*, cost.

We focused on features that allow Bandit to learn if a given VM is suitable for a particular query (e.g., due to memory requirements), which queries could be expected to be long running (e.g., a high number of joins), as well as features correlated with cache behavior. Since analytic queries are often I/O-bound, properly utilizing caching is critical to achieving good performance. Hence, a *cache-aware* service can greatly increase throughput by placing queries with similar physical plans sequentially on the same machine, preventing cache evictions and thrashing.

Finally, we note that these features are appropriate for analytic read-only workloads. We do not intend for these features to be a complete or optimal set. Instead, we intend to demonstrate how even a small set of features that are only weakly related to monetary cost can perform well. Next we describe our features, dividing them into two types, the first related to the incoming query, and the second related to the underlying VM.

Query-related features Our query-related features are extracted from the query plan generated by the database before the query is executed. The features extracted are:

1. **Tables used by current query**: We extract the tables used by the query to allow our model to learn, at a low level of granularity, which queries access the same data and hence could benefit from caching when executed on the same machine.

2. **Number of table scans**: The number of table scans of the query (extracted by the query's execution plan) can help Bandit learn when to anticipate long execution times since table scans are often less efficient than index-based scans.

3. **Number of joins**: Table joins often represent massive increases in cardinalities or time-consuming processes. Thus, the number of joins in a query can be an informative feature.

4. **Number of spill joins**: Spill join operators, which are joins that the query optimizer knows will not fit in memory, must perform disk I/Os due to RAM constraints. This feature helps Bandit learn which queries should be given to VMs with more memory, as well as indicates which queries may have high latency.

5. **Cache reads in query plan**: This feature captures the number of table scan operations that overlap with data currently stored in the cache. This is particularly useful when multiple queries in our template access the same set of tables but with varying physical plans. In this case, the table usage information is no longer sufficient for Bandit to be cache-aware. Combined with the tables used by the current and previous queries, this feature provides substantial information about how a query will interact with the cache.

Virtual machine features Our learning framework also needs to be aware of the resources available on each running VM as well as on available VM configurations. These features help us understand how a particular VM is performing, if there is a "noisy neighbor", etc. These features are collected when a query arrives at a CMAB (the data is collected from the corresponding VM), while another query may still be executing. Specifically, we collect the following features via standard Linux tools:

1. **Memory availability**: This is the amount of RAM currently available in the VM. This helps us understand how RAM pressure from other queries, the operating system, etc. may affect query performance. It also allows us to differentiate between VM types with different amounts of RAM.

2. **I/O rate**: This feature gives the average number of physical (disk) I/Os done per minute over the last query execution. This helps Bandit understand when a machine's storage is performing poorly, as well as giving Bandit a general gauge of the VMs I/O capacity, which may differ even within the same pricing tier.

3. **Number of queries in the queue**: We track the number of queries waiting in each machine's queue. This feature helps Bandit learn when a queue is too full, suggesting that another accepted query would have to wait too long before being processed.

4. **Tables used by last query**: This feature indicates which tables were used by the previous query. This helps Bandit learn which VMs might have useful information in their cache for the current query.

5. **Network cost**: This feature is used when data is partitioned across multiple VMs. In this case, the node that executes the query typically requests necessary data from other nodes. Depending on the query and the distribution of data across the cluster, assigning the query to a different node might incur different network transfer costs. This feature captures the amount of data a node has to move over the network from other nodes in order to process a query. It is roughly estimated by summing the size of all non-local partitions that *may* be required by the query.

3.3 Probabilistic Action Selection

A major challenge of our approach is selecting low-cost actions based on the collected observations. While acceptable results can be achieved with very limited experience, simply *exploiting* this knowledge by repeating "safe" decisions might pass up opportunities for large improvements. Hence, improving the model over time requires the *exploration* of new (potentially high-cost) decisions. Therefore, each CMAB must select actions in a way that addresses this exploration-exploitation dilemma.

One algorithm for effectively solving this problem is Thompson sampling [48], a technique for iteratively choosing actions for the CMAB problem and incorporating feedback. Thompson sampling is well-known in the field of reinforcement learning and has been used for a wide variety of applications including web advertisement, job scheduling, routing, and process control [15, 25]. The basic idea is to choose an arm (action) according to the probability of that particular arm being the best arm given the experience so far. Thompson sampling has been shown to be self-correcting [7] and efficient to implement.

We apply Thompson sampling to each CMAB in the network as follows. Each time a query finishes executing, each CMAB that made a decision related to that query adds to it set of observations $\mathcal{D}$ a new tuple $\{a, x, c\}$, where a is the decision it made, x is the context it used to make that decision, and c is the cost of the decision. Hence, a CMAB's set of experiences $\mathcal{D}$ grows over time.

In order to select actions based on past experience, we assume that there is a likelihood function $P(c|\theta, a, x)$, where θ are the parameters of a model that predicts the cost c for a particular action a given a context x. Given the perfect set of parameters θ^*, this model would exactly predict the cost for any given action and context. Then the problem of selecting the optimal action would be reduced to finding the minimum cost action a where the cost of each action is predicted by this perfect model.

While one clearly cannot know the perfect model (the perfect parameters θ^*) ahead of time, one can *sample* a set of parameters θ' from the distribution of parameters conditioned on past experience, $P(\theta|\mathcal{D})$. Then one can randomly choose an action a according to the probability that a is optimal as follows [15, 48]: sample a set of model parameters θ' from $P(\theta|\mathcal{D})$ and then choose an action that minimizes cost assuming that $\theta' = \theta^*$:

$$\min_{a'} \mathbb{E}(c|a', x, \theta')$$

Conceptually, this means that the system instantiates its beliefs (θ') randomly at each timestep according to $P(\theta|\mathcal{D})$ (i.e., selects a model for predicting the cost based on the probability that the model explains the experience collected so far), and then acts optimally assuming this random model is correct. If one wanted only to exploit existing knowledge, one would not sample from $P(\theta|\mathcal{D})$, but would instead select the mean value of $P(\theta|\mathcal{D})$, a approach that maximizes *exploitation*. On the other hand, choosing a model entirely at random maximizes *exploration*. Instead, the Thompson sampling approach (drawing θ from $P(\theta|\mathcal{D})$) balances exploration and exploitation [7, 44].

Using Thompson sampling in the context of cloud computing is extremely advantageous. Traditional techniques must accurately model many complex systems: virtual machines hosted on cloud infrastructures can exhibit erratic behavior when load is high; optimizers in modern databases may use probabilistic plan generation techniques, potentially creating variance in how identical queries are processed; query execution engines can exhibit sporadic behavior from cache misses, context switches, or interrupts. Our approach deals with complexity across the entire cloud environment end-to-end by modeling the relationship between various context features and cost *probabilistically*. When an action has an unexpectedly low or high cost, we do not need to diagnose which component (the VM, the optimizer, the execution engine, the hardware itself) is responsible. We can simply add the relevant context, action, and cost to the experience set. If the unexpected observation was an one-off outlier, it will not have a significant effect on the sampled models. If the unexpected observation was indicative of a pattern, Thompson sampling ensures that the pattern will be properly explored and exploited over time.

Regression trees Bandit uses REP trees (regression trees) [27] to model the cost of each potential action in terms of the context. The parameter set θ represents the splits of a particular tree model, i.e. the decision made at each non-leaf node. To use REP trees with Thompson sampling, we need a way to sample a regression tree based on our past experience (in other words, to sample a θ from $P(\theta|\mathcal{D})$). Since generating every possible regression tree would be prohibitively expensive (there are $O(n^n)$ possible trees), we utilize *bootstrapping* [13]. In order to sample from $P(\theta|\mathcal{D})$, we select $n = |\mathcal{D}|$ tuples from $\mathcal{D}$ *with replacement* (so the same tuple may be selected multiple times or not at all) to use as a training set for the regression tree learner. Bootstrapping has been shown to accurately produce samples from $P(\theta|\mathcal{D})$ [21]. In short, this is because there is a non-zero chance that the entire sampled training set will be composed of a single experience tuple (full exploration), and the mean of the sampled training set is exactly $\mathcal{D}$ (full exploitation).

We choose REP trees because of their speed and ease of use, but any sufficiently powerful modeling technique (neural networks, SVR, etc.) could be used instead.

Action independence Reinforcement learning and Thompson sampling literature has traditionally treated each arm of the CMAB as independent random variables, and has also assumed that the next context observed is independent of the action taken. Although neither of these conditions hold here, we demonstrate later that algorithms designed to solve the CMAB problem work well in our context. This is not surprising, as independence assumptions are often successfully ignored when applying machine learning techniques to real world problems like natural language processing [33] and, in the specific case of Thompson sampling, web advertising and route planning [25].

Bounding the strategy space Since our action selection algorithm must balance exploration and exploitation, it may consider high-cost strategies, especially when little information is available. To reduce such catastrophic "learning experiences," Bandit uses a heuristic search algorithm for limiting its search space: it forbids picking the `Accept` option when a machine's queue has more than b queries in it. From the remaining actions, each CMAB picks the one that is expected to minimize the cost. This technique has previously been called *beam search* [35].

Setting the b threshold is the responsibility of the application, but for many SLOs a good threshold can be calculated. For example, if the SLO requires that no query takes longer than x minutes, we can set $b = \frac{x}{q_{min}}$, where q_{min} is a lower bound on query execution time. This prevents Bandit from placing too many short queries on the same queue. The violations would be even worse if one considers longer running queries. We note that even without beam search, Bandit will eventually learn that no more than b queries should be placed in the queue at once, but eliminating these options *a priori* accelerates convergence. However, one must be careful not to set b too low, which could eliminate viable strategies and cause Bandit to converge to a local optima.

Placing more constraints on the strategy space may also decrease the time required to converge to a good strategy. For example, we prevent VMs with no queries in their queues from selecting the `Pass` arm. This prevents provisioning multiple VMs to process a single query. It is worth noting that while such restrictions may accelerate convergence, they are not needed: Bandit will still converge to a good strategy without them. As with the b value, one should be wary of limited the strategy space too much, as one could unknowingly eliminate a good strategy.

Experience size Since the experience $\mathcal{D}$ of each CMAB consists of action/context/cost triples (a, x, c), and since a new triple is added to $\mathcal{D}$ on a number of CMABs each time a query completes, one may be concerned with the memory usage of the experience array itself. Even though each experience tuple (as described here) could be represented using relatively little space (encoding the cost, action, and each feature as a 32-bit integer requires only 448 bits per experience tuple), the system will continue to use more memory as long as it continues to process queries. However, since query workloads tend to shift over time, newer experiences are more likely to pertain to the current environment than older ones. One solution to this problem could be to bound the size of the experience set, and remove the oldest experiences when new experiences arrive, or one could probabilistically decrease the weights of older experiences, eventually removing them when they no longer have a significant effect on performance [12, 26]. Both approaches have shown good performance in real-world applications [25].

3.4 Releasing Resources

So far, we have discussed provisioning new VMs and assigning queries to VMs, but we have not investigated shutting down VMs. Since a cloud-deployed database application must pay for machines until they are turned off, deciding when to release a machine is important. Previous works [10, 24, 32, 37] have simply shutdown a machine when that machine had no more queries in its processing queue. While simple, this strategy can be disastrous when a query arrives just after the previous query finishes; in this case, the machine that was just released must be re-provisioned, and the cost of initializing the VM must be paid again.

If the arrival time of the next query to be served on a particular VM was known ahead of time, then one could simply calculate if it would be cheaper to keep the VM running until the next query arrives or to shut down and restart the machine. Of course, in general, this is not possible. One might try to keep a machine active for some constant number of seconds k after its queue empties, and then shut the machine down if it still has not received a query to process. This approach works well when the query arrival rate is a specific constant, but performs poorly in general.

Hill-climbing method In order to determine whether to shut down a machine or keep it running in anticipation of a future task, we developed a hill-climbing based learning approach. Each machine maintains and adjusts a variable k, which represents the number of seconds to wait once the machine is idle before shutting down. For all machines, we initialize $k = 1sec.$ If no query arrives after k seconds, the machine shuts down. If another query arrives before k seconds have passed, the machine processes that query and remains online.

We then adjust the wait period k as follows: we determine if it was profitable to wait for this query to arrive, or if it would have been better to shut the machine down and restart it. If the latter decision would have been more profitable, we reduce the wait period to $k' = k/\lambda$, where λ is our learning rate, described below. If the next query arrives after the machine has been shut down, and we determine that it would have been more profitable to have kept the machine

running, then we increase the wait period to $k' = k \times \lambda$.

Here, $\lambda > 1$ is the *learning rate* of the algorithm, which represents how much new information is valued over past experience. Setting λ closer to one causes k to adjust itself slowly (far-sighted), whereas setting λ to a high value causes k to be adjusted quickly (near-sighted). While a system can benefit from tuning λ, we find that $\lambda = 2$ works well in many situations where query arrival rates match real-world data.

Alternative learning methods While we have experimented with the hill climbing approach described above, any reinforcement learning algorithm with a continuous action space could be applied to learn the wait period before shutting down a VM. Examples of such algorithms include Monte Carlo-based approaches [9], continuous Q-learning [23], and the `HEDGER` algorithm [46]. Each of these algorithms can be applied to select the next wait time after a machine becomes idle. Since a cost can be generated for each decision (after another query is accepted by that VM), any contextual continuous action space approach could be applied.

4. EXPERIMENTS

We implemented Bandit and tested its effectiveness and training overhead on Amazon EC2 [2], using three types of machines: `t2.large`, `t2.medium`, and `t2.small`. In the majority of our experiments, we used workloads generated from TPC-H [6] templates. However, we include experiments (Section 4.2) with a larger set of templates extracted from Vertica's [30] performance testing suite. Unless otherwise stated, all queries were executed on a 10GB database stored in Postgres [5], and each VM holds its own complete copy of the entire database, i.e. a fully-replicated database.

We model query arrival as a non-homogenous Poisson process where the rate is normally-distributed with constant mean arrival rate of 900 queries per hour and variance $k = 2.5$, which is representative of real-world workloads [34]. Our experiments measure the average cost of each executed query, and each point plotted represents a sliding window of 100 queries. We present the most representative results of our experimental study, aiming to illustrate the effectiveness and capabilities of reinforcement learning systems when applied to resource and workload management.

Feature Extraction Our virtual machine features are extracted using via standard Linux tools. The query features are extracted by parsing the query execution plan. One challenge we faced was calculate the number of spill joins, i.e., the joins for which the query optimizer predicts that they will not fit in memory. Computing the exact number of spill joins in the query plan may depend upon accurate cardinality estimations for a specific query. We calculate the number of spill joins in a query plan in a very relaxed way: a join of table T_1 and table T_2 is considered to be a spill join if and only if the maximum possible size of the result exceeds the amount of RAM (i.e., the total size of T_1 times the total size of T_2 exceeds the size of RAM), regardless of the join predicate involved. While this is a conservative estimation of which joins will spill, our estimate still has *some* meaningful relationship with query execution cost as discussed in Section 4.1.

4.1 Effectiveness

Next, we demonstrate the effectiveness of our learning approach and feature set to generate low-cost solutions for deploying data management applications on IaaS clouds.

Multiple SLO Types We evaluated Bandit's ability to enforce four commonly used SLO types [16, 17, 34]: (1) `Average`, which sets an upper limit (of 2.5 times the average latency of each query template in isolation) on the average query latency so far, (2) `Per Query`, which requires that each query completes within a constant multiple (2.5) of its latency, (3) `Max`, which sets an upper limit on the latency of the whole workload (2.5 times the latency of the longest query template), and (4) `Percentile`, which requires that no more than 10% of the queries executed so far exceed a limit (2.5 times the average latency of the query templates in isolation). We assume the monetary cost for violating the SLO is one cent per second.

We compared Bandit against the *optimal* strategy for these SLO types. Specifically, we generated a sequence of thirty queries drawn randomly from TPC-H templates and we brute forced the optimal decisions (VMs to rent and query placement on them) to process this sequence with minimal cost. We then trained Bandit on this workload by repeating this sequence many times, allowing all thirty queries to finish before submitting the next sequence[2], until its cost converged. We compared Bandit to a *clairvoyant greedy* strategy, which uses a perfectly accurate latency model to estimate the cost of each decision, and, upon the arrival of a new query, makes the lowest-cost decision [32]. Finally, we used a simple round-robin scheduling strategy to divide the thirty queries across seven VMs (the number of VMs used in the optimal solution). Figure 3a shows this comparison for the four SLO types.

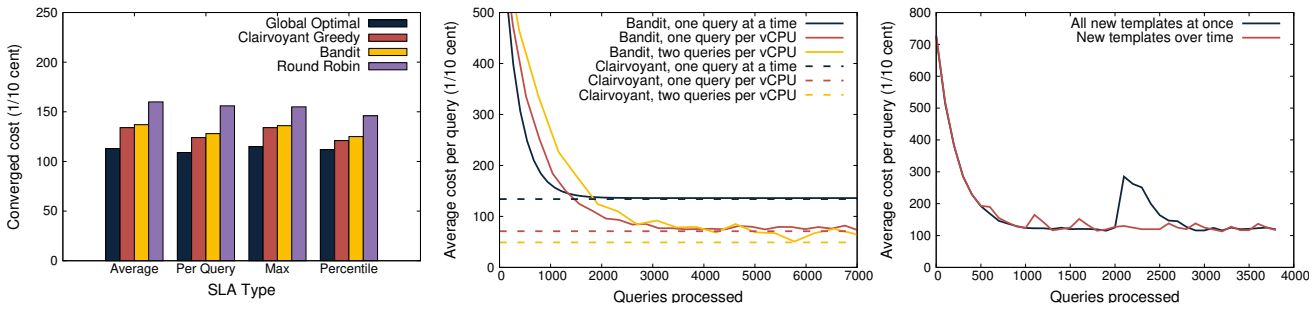
The results are very positive. Bandit achieves a final cost ranging from 8% to 18% of the global optimum; but computing the optimal solution requires both a perfect latency model and a significant amount of computing time (in some cases the problem is `NP-Hard`). Bandit also represents a significant cost reduction over naive, round-robin placement. Finally, Bandit's model comes within 4% of the clairvoyant greedy model. *This means that the cost model developed by Bandit– which only implicitly models query latency – can perform at almost the same level as an approach with a perfect latency prediction model.*

Concurrent Queries Bandit is able to converge to effective models when queries execute concurrently, when performance prediction is quite challenging. Figure 3b shows the convergence of the cost per query over time for Bandit for various concurrency levels with a `Max` SLO. Here, queries are drawn randomly from TPC-H templates and their arrival time is drawn from a Poisson process. *One query* represents no concurrent executions, i.e., we admit only one query at a time on each machine. *One query/vCPU* and *Two queries/vCPU* represent running up to one or two queries respectively per virtual CPU core on each machine. In the *two queries/vCPU* case, `t2.small` machines run two queries at once, and `t2.medium` and `t2.large` machines run four queries at once.[3]

The results show that increased concurrency levels incur more training overhead (convergence takes longer), but a lower converged cost since the cost-per-VM-hour is the same regardless of how many CPU cores are utilized. Since identifying the optimal strategy for these scenarios is not

[2]This lowered the average query arrival rate from 900 queries/hour to 200 queries/hour.

[3]Each query is itself executed serially. In other words, there is parallelism between queries, but not within queries.



(a) Effectiveness of different heuristics (b) Learning on concurrent queries (c) Learning on unseen queries

Figure 3: Effectiveness of Bandit for various scenarios.

straightforward, we compare Bandit's performance against a clairvoyant greedy strategy. Again, Bandit performs within 4% of the clairvoyant greedy strategy. *Hence, Bandit's direct cost modeling approach handles high levels of concurrency with no pre-training.* Both Bandit and the clairvoyant greedy strategy utilized fewer VMs at increased concurrency levels. With no concurrency, both strategies used an average of 45 VMs. With one or two queries per vCPU, both strategies used an average of 38 VMs.

Adaptivity to new templates Handling previously-unseen queries represents an extreme weakness of pre-trained query latency prediction models. Bandit can efficiently handle these cases. Figure 3c shows cost curves for two different scenarios. In both scenarios, Bandit begins processing a workload consisting of queries drawn randomly from 13 TPC-H templates, with the performance goal set to the `Max` SLO type we defined above. In the `all new templates at once` scenario, seven new query templates are introduced after the 2000th query has been processed. In the `new templates over time` scenario, a new query template is introduced every 500 queries. Introducing seven new query templates at once causes a notable increase in cost. Bandit eventually recovers as it gains information about the new query templates. However, introducing queries slowly over time causes only a slight decrease in Bandit's performance, and Bandit recovers from the small change faster than it did for the large change. This makes Bandit especially well-suited for query workloads that change slowly over time.

4.2 Convergence & Training Overhead

The three plots in Figure 4 show convergence curves for Bandit in different scenarios. Each curve shows the average cost per query for a sliding window of 100 queries compared to the number of queries processed.

Impact of SLA strictness Figure 4a shows the convergence curve for various SLA strictness levels for the `Max` SLA type where the deadline for each query is set to 1.5, 2.5, and 3.5 times the latency of that query in isolation. Looser SLAs take longer to converge, but converge to a lower value. Tighter SLAs converge faster, but have higher average cost. This is because looser SLAs have a larger policy space that must be explored (there are more options that do not lead to massive SLA violation penalties), whereas tighter SLAs have smaller policy spaces. Intuitively, this is because any strategy that does not violate a strict SLA will not violate a looser SLA either.

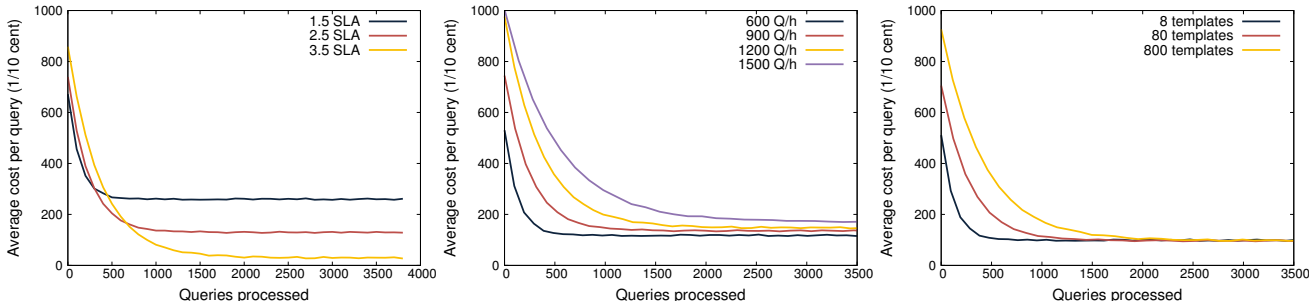
Impact of arrival rate Figure 4b shows convergence curves for Bandit for various query arrival rates. The graph matches an intuitive notion that high query arrival rates should be more difficult to handle than low query arrival rates. Higher query arrival rates require more complex workload management strategies that take Bandit longer to discover. For example, with a low query arrival rate, Bandit may be able to assign all queries using a particular table to a single VM, but with a high query arrival rate, Bandit may have to figure out how to distribute these queries across several machines.

Impact of query templates Since TPC-H provides only a small number of query templates, we also evaluated Bandit's performance on 800 query templates extracted from Vertica's [30] analytic workload performance testing suite. These templates are used to measure the "across the board" performance of the Vertica database, and thus they cover a extensive variety of query types. The generated queries are ran against a 40GB database constructed from real-world data. For consistency, we still use Postgres to store and query the data. Figure 4c shows convergence curves for randomly generated workloads composed of 8, 80, and 800 query templates. For the 8 template run, we selected the four query templates with the highest and lowest costs (similarly, for the 80 query template run, we selected the 40 query templates with the highest and the lowest cost) so that the average query cost is the same for all three runs.

Higher template counts take longer to converge since the corresponding strategy space is larger. Workloads with fewer query templates exhibit less diversity, and Bandit is able to learn an effective strategy faster. Even when the template count is very large (800), Bandit still finds good strategies after having seen a reasonable number of queries.

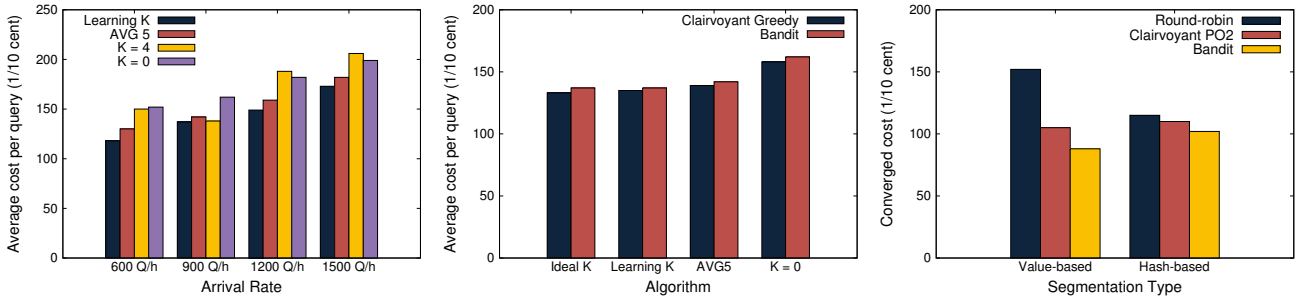
4.3 Shutdown strategy

Impact of query arrival rate After a VM is provisioned, Bandit must decide when to turn it off. Figure 5a compares different shutdown strategies for various query arrival rates. A constant delay of $K = 4$ (wait four seconds for a new query once the queue is empty) can be very effective for certain arrival rates (900 Q/h), but will not perform well for others (1200 Q/h, 1500 Q/h). `Learning K` represents the algorithm described in Section 3.4. `AVG5` sets the time to wait before shutting down a VM to the average ideal wait time of the last 5 shutdown decisions we had to make. This is calculated as follows: after deciding to keep a machine on or off we compute what would have been the ideal delay and



(a) Convergence vs. SLA strictness level (b) Convergence vs. query arrival rates (c) Convergence vs. number of templates

Figure 4: Convergence behavior of Bandit for various scenarios.



(a) Shutdown strategy, varying arrival rate (b) Bandit compared to clairvoyant (c) Value vs. hash based segmentation

Figure 5: Shutdown strategies and partitioning schemes

set the new delay to the average of the last five ideal delay values we computed. `K=0` represents shutting down a VM immediately after its queue becomes empty. Figure 5a shows that `Learning K` is the best practical strategy independent of the arrival rate. The increase in reward for `Learning K` are larger when the query arrival rate is slower. This is because slower arrival rates cause VM queues to empty more frequently, making the decision on whether to shutdown a machine or to keep it running (in anticipation of another query) more important.

Comparison with clairvoyant We also compare Bandit to the clairvoyant greedy strategy using each of the shutdown strategies described above for a query arrival rate of 900 queries per hour. Figure 5b shows the comparison. In this experiment, we additionally compare to the *ideal* shutdown strategy. This strategy works by "looking into the future" and retroactively making the correct decision about whether or not to shutdown a machine based on if a query will be placed onto it again in a profitable timeframe. This is done by iteratively running queries, noting the arrival time of of the n-th query on each machine, and then restarting the process (this time running to the $(n+1)$-th query). This represents the optimal shutdown policy, given a particular strategy. Clearly, this is impossible in practice. However, our `Learning K` performs within $1-3\%$ of this optimal.

4.4 Partitioned Datasets

We have thus far limited our experiments to fully-replicated database systems in which any machine is capable of executing any query independently. While such systems have many applications, modern analytic databases typically partition data across multiple machines. We experimented with such scenarios. Next, we discuss these results.

For this experiment, we used a cloud-deployment of a commercial analytic column store database. We generated 20TB of TPC-H data and loaded it into the commercial engine deployed on AWS. We partitioned the large fact table (`lineitem`) and replicated the other tables. We used two different partitioning scheme: *value-based partitioning*, in which a given partition will store tuples that have attribute values within a certain range, and *hash-based partitioning*, which partitions tuples based on a hash function applied on a given attribute. Each partition is also replicated by a factor of $k = 2$.

For Bandit, we initialized a cluster with three VMs at the start of each experiment and the partitions assigned to each VM is determined by the underlying commercial database engine we used. For both partitioning schemes, we compare Bandit with two different techniques:

1. A `Round-Robin` approach, which uses a fixed cluster size of $n = 21$ VMs and dispatches incoming queries to these VM in a circular order. We evaluated all cluster sizes from $n = 1$ to $n = 50$, and selected $n = 21$ because it had the best performance for our workload.

2. A clairvoyant power of two method [38, 42] (labeled as `Clairvoyant PO2`), which randomly selects two machines from the current pool and schedules the query on whichever of those two machines will produce the lowest cost (as determined by a clairvoyant cost model). Among the available machine the algorithm can choose from we include three "dummy VMs", which represent provisioning new VMs of the three EC2 types we used.

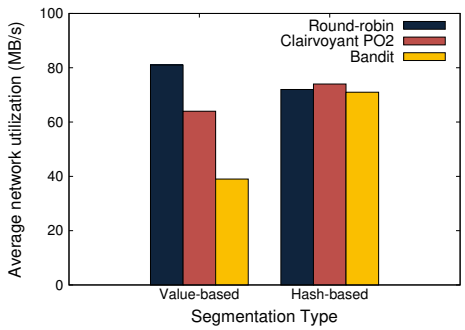


Figure 6: Network utilization

In these experiments we used the `Max` SLO metric. Figure 5c shows the converged cost for each partitioning scheme. For the hash-based partitioning, Bandit outperforms the Clairvoyant PO2 method by a small margin. Hash-based partitioning allows for each node in the cluster to participate evenly (on average) in the processing of a particular query, as tuples are distributed evenly (on average) across nodes. Indeed, Bandit learns a basic round-robin query placement strategy, but still intelligently maintains the right size of the cluster, in contrast to the round-robin method that has a static cluster size.

Bandit outperforms Clairvoyant PO2 more significantly when using value-based partitioning (16%). This partitioning schema allows for data transfer operations to be concentrated on the particular nodes that have the range values required by the query. In this case, Bandit learns to maximize locality, i.e., it learns to place certain queries on nodes that have most (or all) the partitions necessary to process a certain query. Generally, Bandit learns to assign each query to a node that will incur low network cost, which leads to reduced query execution time and hence lower monetary cost.

Note that the round-robin technique performs significantly worse with value-based partitioning than with hash-based partitioning, because an arbitrarily selected node is less likely to have as much data relevant to the query locally with value-based partitioning than with hash-based partitioning, causing more data to be sent over the network.

Figure 6 shows the average total network utilization during the experiment. It verifies that networking overhead is a substantial factor in the performance differences shown in Figure 5c. For hash-based partitioning, the network utilization is approximately equal for all three methods, but for value-based partitioning, Bandit requires substantially less data transfer. Generally, value-based partitioning needs to be carefully configured by a DBA, whereas hash-based methods are more "plug-and-play". However, these results show that value-based partitioning can lead to greatly reduced network utilization when combined with an intelligent workload management system.

5. RELATED WORK

Other approaches to online query scheduling, like the Sparrow [42] and other "Power of Two" [38] schedulers, seek to minimize scheduling latency for time-sensitive tasks. They achieve results significantly better than random assignment by sampling two potential servers and assigning the query to whichever server is "best", as determined by a heuristic. While such approaches achieve excellent latency, they often depend on latency prediction models and do not handle cluster sizing/resource provisioning. The clairvoyant greedy algorithm presented in our experiments is equivalent to a "Power of N" technique, where every server is considered and the best is selected via a clairvoyant cost model. While this is impossible in practice, we have demonstrated that Bandit performs similarly to this "Power of N" technique.

When query latencies are constant (e.g. have the same latency regardless of cache, machine type, etc) and known at query arrival time, the problem of workload management under a `Max` SLA is isomorphic to the online bin packing problem, for which there are known heuristics with asymptotic performance ratios [45]. While this relaxation is attractive for many reasons, it is difficult to actualize due to the complexity of ahead-of-time latency prediction. Additionally, ignoring cache and different machine tiers can drastically affect performance and cost. Bandit avoids the dependency on latency prediction models and takes advantage of different machine tiers and caches.

The problem of finding sensical service level agreements has been previously examined [40]. Here, the focus is not on finding a good strategy given a performance constraint, but to find performance constraints that illustrate performance vs. cost trade-offs in cloud systems. Recently, this system has been expanded [41] to include a reinforcement learning approach to cluster-scaling which probabilistically meets performance goals.

The SmartSLA [52] system looks at how to divide up the resources (e.g., CPU, RAM) of a physical server among multiple tenants to minimize SLA penalties in a DBaaS (database as a service) setting. They use machine learning models to predict SLA violation costs for various proposed resource distributions in order to minimize costs *for the cloud provider*. SmartSLA takes a fine-grained approach by managing resource shares, but leaves cluster sizing and query scheduling decisions to the underlying database software. Bandit treats each VM as an indivisible resource, but additionally makes scheduling and cluster-sizing decisions. Further, Bandit seeks to minimize the user's cost, not the cloud provider's cost. Other works that have focused on the lowering the cloud provider's cost focus on co-locating tenants advantageously, either by minimizing the number of servers provisioned [18], maintaining a certain number of transactions per second [31], or maximizing the profit of the cloud provider [34].

As mentioned in the introduction, many previous works have addressed the problem of resource provisioning [43,47], query placement [14, 22, 28, 29, 31, 34, 36], query scheduling, [16, 17, 42] and admission control [49, 51] for only a subset of the SLAs supported by Bandit. Works that handle many different types of SLAs or spanned more than one of these tasks [10, 24, 32, 37] have all depended on explicit latency prediction models, a notoriously difficult problem for cloud environments [11, 39].

6. CONCLUSIONS

Fully realizing the promise of elastic cloud computing will require a substantial shift in how we deploy data management applications on cloud infrastructure. Current applications focus too heavily on manual, human-triggered scaling which is too slow to respond to rapid increases or decreases in demand. Failing to spin up more resources when they are

needed or renting resources for longer than required leads to degraded performance, wasted resources, and potentially substantial monetary costs.

Existing research on these challenges leans too heavily on explicit query latency prediction models, which become inaccurate due to noisy neighbors, high levels of concurrency, and previously unseen queries. While convenient and even highly accurate for single-node database systems, approaches based on latency prediction are not easily retrofitted for cloud environments.

We argue that there is significant space for new research that applies machine learning techniques to address workload and resource management challenges for cloud databases. As a proof-of-concept, we have presented Bandit, a cloud service that uses reinforcement learning techniques to learn, over time, low cost resource provisioning and query scheduling strategies. Bandit is able to adapt and continuously learn from shifting workloads while remaining resilient to variance caused by concurrency.

While putting scalability decisions in the hands of machine learning algorithms may be uncomfortable for some, cloud systems are not going to become any simpler. Hence, we strongly believe that the ever-increasing diversity of options offered by IaaS providers will only increase the need for end-to-end, machine learning based approaches to handle existing and future challenges faced by data management applications.

7. ACKNOWLEDGEMENTS

This research was funded by NSF IIS 1253196.

8. REFERENCES

[1] Amazon RDS, https://aws.amazon.com/rds/.
[2] Amazon Web Services, http://aws.amazon.com/.
[3] Google Cloud Platform, https://cloud.google.com/.
[4] Microsoft Azure Services, http://www.microsoft.com/azure/.
[5] PostgreSQL database, http://www.postgresql.org/.
[6] The TPC-H benchmark, http://www.tpc.org/tpch/.
[7] S. Agrawal et al. Further optimal regret bounds for Thompson sampling. In *AISTATS '13*.
[8] M. Akdere et al. Learning-based query performance modeling and prediction. In *ICDE '12*.
[9] Aless et al. Reinforcement learning in continuous action spaces through sequential Monte Carlo methods. In *NIPS '07*.
[10] Y. Azar et al. Cloud scheduling with setup cost. In *SPAA '13*.
[11] S. K. Barker et al. Empirical evaluation of latency-sensitive application performance in the cloud. In *MMSys '10*.
[12] O. Besbes et al. Stochastic multi-armed-bandit problem with non-stationary rewards. In *NIPS '14*.
[13] L. Breiman. Bagging predictors. In *Maching Learning '96*.
[14] U. V. Catalyurek et al. Integrated data placement and task assignment for scientific workflows in clouds. In *DIDC '11*.
[15] O. Chapelle et al. An empirical evaluation of Thompson sampling. In *NIPS'11*.
[16] Y. Chi et al. iCBS: Incremental cost-based scheduling under piecewise linear SLAs. *PVLDB '11*.
[17] Y. Chi et al. SLA-tree: A framework for efficiently supporting SLA-based decisions in cloud computing. In *EDBT '11*.
[18] C. Curino et al. Workload-aware database monitoring and consolidation. In *SIGMOD '11*.
[19] J. Duggan et al. Contender: A resource modeling approach for concurrent query performance prediction. In *EDBT '14*.
[20] J. Duggan et al. Performance prediction for concurrent database workloads. In *SIGMOD '11*.
[21] B. Efron. Better bootstrap confidence intervals. *American Statistical Association '87*.
[22] A. J. Elmore et al. Characterizing tenant behavior for placement and crisis mitigation in multitenant DBMSs. In *SIGMOD '13*.
[23] C. Gaskett et al. Q-learning in continuous state and action spaces. In *AKCAI '99*.
[24] S. Genaud et al. Cost-wait trade-offs in client-side resource provisioning with elastic clouds. In *CLOUD '11*.
[25] A. Gopalan et al. Thompson sampling for complex online problems. In *ICML '14*.
[26] N. Gupta et al. Thompson sampling for dynamic multi-armed bandits. In *ICMLA '11*.
[27] M. Hall et al. The WEKA data mining software: An update. *SIGKDD '09*.
[28] E. Hwang et al. Minimizing cost of virtual machines for deadline-constrained MapReduce applications in the cloud. In *GRID '12*.
[29] V. Jalaparti et al. Bridging the tenant-provider gap in cloud services. In *SoCC '12*.
[30] A. Lamb et al. The Vertica analytic database: C-store 7 years later. *PVLDB '12*.
[31] W. Lang et al. Towards multi-tenant performance SLOs. In *ICDE '14*.
[32] P. Leitner et al. Cost-efficient and application SLA-aware client side request scheduling in an IaaS cloud. In *CLOUD '12*.
[33] D. D. Lewis. Naive (Bayes) at forty: The independence assumption in information retrieval. In *ECML '98*.
[34] Z. Liu et al. PMAX: Tenant placement in multitenant databases for profit maximization. In *EDBT '13*.
[35] B. T. Lowerre. *The Harpy Speech Recognition System.* PhD thesis, Stanford University, 1976.
[36] H. Mahmoud et al. CloudOptimizer: Multi-tenancy for I/O-bound OLAP workloads. In *EDBT '13*.
[37] R. Marcus et al. WiSeDB: A learning-based workload management advisor for cloud databases. *PVLDB '16*.
[38] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Parallel Distrib. Sys. '01*.
[39] V. Narasayya et al. SQLVM: Performance isolation in multi-tenant relational database-as-a-service. In *CIDR '13*.
[40] J. Ortiz et al. Changing the face of database cloud services with personalized service level agreements. In *CIDR '15*.
[41] J. Ortiz et al. PerfEnforce demonstration: Data analytics with performance guarantees. In *SIGMOD '16*.
[42] K. Ousterhout et al. Sparrow: Distributed, low latency scheduling. In *SOSP '13*.
[43] J. Rogers et al. A generic auto-provisioning framework for cloud databases. In *ICDEW '10*.
[44] D. Russo et al. An information-theoretic analysis of Thompson sampling. *Machine Learning Research '14*.
[45] S. Seiden. On the online bin packing problem. *JACM '02*.
[46] W. Smart et al. Practical reinforcement learning in continuous spaces. In *ICML '00*.
[47] B. Sotomayor et al. Virtual infrastructure management in private and hybrid clouds. *IEEE IC '09*.
[48] W. R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika '33*.
[49] S. Tozer et al. Q-Cop: Avoiding bad query mixes to minimize client timeouts under heavy loads. In *ICDE '10*.
[50] S. Venkataraman et al. Ernest: efficient performance prediction for large-scale advanced analytics. In *NSDI '16*.
[51] P. Xiong et al. ActiveSLA: A profit-oriented admission control framework for Database-as-a-Service providers. In *SoCC '11*.
[52] P. Xiong et al. Intelligent management of virtualized resources for database systems in cloud environment. In *ICDE '11*.