A Generic Auto-Provisioning Framework for Cloud Databases

Jennie Rogers [1], Olga Papaemmanouil [2], Ugur Cetintemel [1]
[1]*Brown University,* [2] *Brandeis University*
jennie@cs.brown.edu, olga@cs.brandeis.edu, ugur@cs.brown.edu

***Abstract*— We discuss the problem of resource provisioning for database management systems operating on top of an Infrastructure-As-A-Service (IaaS) cloud. To solve this problem, we describe an extensible framework that, given a target query workload, continually optimizes the system's operational cost, estimated based on the IaaS provider's pricing model, while satisfying QoS expectations. Specifically, we describe two different approaches, a "white-box" approach that uses a fine-grained estimation of the expected resource consumption for a workload, and a "black-box" approach that relies on coarse-grained profiling to characterize the workload's end-to-end performance across various cloud resources. We formalize both approaches as a constraint programming problem and use a generic constraint solver to efficiently tackle them.**

We present preliminary experimental numbers, obtained by running TPC-H queries with PostsgreSQL on Amazon's EC2, that provide evidence of the feasibility and utility of our approaches. We also briefly discuss the pertinent challenges and directions of on-going research.

I. Introduction

Cloud computing is widely touted as "the greatest thing since sliced bread", as it allows computing to behave much like a generic utility that is always available on-demand from anywhere. This utility-oriented model of hardware and software usage establishes cloud computing as a potentially transformative technology, while the associated pay-as-you-go model often renders it as a savvy economic choice for many entities that would like to avoid up-front capital expenses and reduce cost of ownership over time.

DBMSs are good candidates for adoption in the cloud — they are difficult to configure and manage and serve workloads that are resource-intensive and highly time varying. While many DBMSs have already been ported as appliances on virtualized hardware offered by IaaS providers, key questions that challenge widespread, cost-effective deployment of cloud databases remain open.

One such question involves database system optimization, which traditionally assumes the availability of a fixed resource configuration and exclusively deals with performance optimization. This view ignores the operational (monetary) cost of running the database. The pay-as-you-go model of cloud computing, however, requires operational costs to be treated as a prime consideration along with performance.

In this paper, we discuss the problem of minimizing the operational cost of running a DBMS on IaaS. The operational cost of a DBMS depends on the monetary cost function of the underlying IaaS provider. That is to say, by how much the provider charges for each resource type. We propose a resource provisioning framework that identifies a collection of minimum-cost infrastructure resources (i.e., a set of potentially heterogeneous virtual machines) that can collectively satisfy a predicted time-varying workload within target QoS expectations. Moreover, at run-time, we make best use of the reserved resources by intelligently routing incoming queries to specific machines.

In this work-in-progress paper, we describe two solutions to the resource provisioning problem. *Black-box* provisioning uses end-to-end performance results of sample query executions, whereas *white-box* provisioning uses a finer grained approach that relies on the DBMS optimizer to predict the physical resource (i.e., I/O, memory, CPU) consumption for each query. We formulate both solutions as multi-dimensional bin-packing problems and efficiently tackle them using a generic constraint solver. We discuss the strengths and limitations of these approaches and present preliminary experimental numbers (based on TPC-H queries running on top of PostgreSQL/Amazon EC2) that offer additional insight about their feasibility and utility.

II. System Model

Cloud environment. Our model assumes a cloud infrastructure similar to the offerings from various IaaS providers [1], [10]. Vendors provide access to pre-configured computing virtual machines (VMs or cloud servers) on which users can remotely install and run their software. Available VM types are differentiated by the resources they provide such as CPU power, I/O bandwidth, disk space and memory size.

Cloud resources are offered through a "pay-as-go" pricing model; they are rentable for a certain *reservation period*, typically an hour, and this *per-period* cost is fixed in that it depends only on the server configuration and not on the actual server utilization over the course of the period. IaaS clouds also provide storage and data transfer services, usually with additional *per request* charges. For example, Amazon's EBS service offers persistent storage volumes and charges per request for each I/O operation. In this work we consider the charges only for the VMs and I/O operations.

Regardless of the pricing model used, it is the responsibility of the IaaS client to provision its resources appropriately. Based on an estimation of the expected workload, clients will deploy, observe, and accordingly adjust their computing

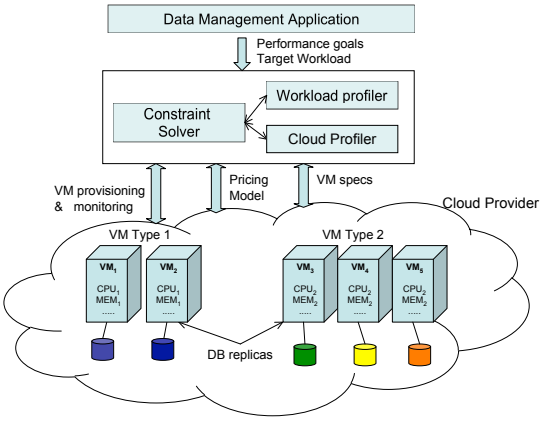

Fig. 1. High-level cloud DBMS and provisioning model.

capacity by reserving or releasing VMs over time to meet any QoS requirements. Our framework addresses this problem.

Database deployment model. Cloud-based DBMSs are typically instantiated through a *database appliance* [3], i.e., a VM image with a pre-installed pre-configured database engine. We assume a replicated database model on top of a shared-nothing network of multiple VMs, each running an independent database appliance (Figure 1). We consider read-mostly workloads common to analytical processing applications. We assume that write operations are performed off-line and in parallel across all our cloud servers.

Target workload. For provisioning to be meaningful, the query workload has to exhibit some predictability. We assume the availability of a representative workload, which can be obtained either manually (e.g., by the application provider or the administrators) or automatically (through DBMS profilers). We require that the representative workload consists of a set of query classes along with the expected distribution of queries across the different classes. In either case, the elasticity of cloud resources allow quick re-provisioning, which ameliorates the negative impact of inaccurate workload predictions.

Resource Management. Given the target workload, QoS goals (expressed as latency bounds), the cloud's resource specification and its monetary cost model, our framework identifies the most cost-effective VM allocations that would meet the QoS expectations of the workload. We propose two provisioning approaches that rely on profiling techniques for evaluating the resource availability of the cloud server, the resource consumption of the target workload, as well as its expected performance and operational cost. The recommended set of VMs are reserved for the next reservation period and each new VM will host a different replica of our database. This process is then repeated for each reservation period, scaling our system up or down depending on the expected workload.

Workload Management. During run-time, our framework routes incoming user queries to the reserved machines. The goal is to assign queries to machines with sufficient resources to execute both new and existing queries within their QoS bounds, while minimizing any extra monetary cost due to per request charges. Moreover, the online query routing must respect any assumptions/decisions of the provisioning phase

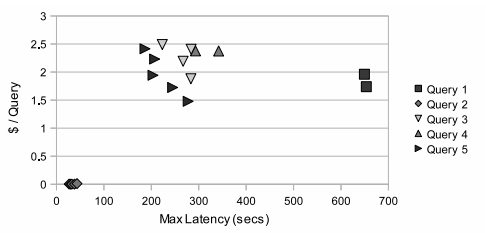

Fig. 2. Latency vs. cost for various input rates on one instance.

(e.g., assignment of certain query types to certain machines, maximum acceptable query rate of each machines).

Constraint Solver. The constraint solver is responsible for solving both the offline resource provisioning and online query routing optimization problems. Both problems are expressed as constraint programs, which get solved by a central coordinator using an off-the-shelf optimization tool.

III. Experimental Configuration

Throughout this document we reference results using a proof-of-concept for our framework deployed on Amazon Web Services [1]. We deployed PostgreSQL 8.4.1 on a variety of AWS instances rated in terms of being I/O-intensive, CPU-intensive or well-weighted for both. We created our workload based on TPC-H Query Classes 1-5 (scale factor 10), which represent a mix of daily (short) and long term (strategic) queries. We also extended the PostgreSQL optimizer to produce estimates for CPU and I/O operations used by each query. We correlated this with actual resources consumed by using kernel-level instrumentation (`procfs, iostat`). To support consistency and high availability in our configuration, we use EBS volumes as storage disks. This creates an additional charge for each I/O operation.

We experimented with two provisioning approaches. Black-box relies on end-to-end query performance and assigns queries to individual instances based on query arrival rates, while white-box provisions a set of VMs using optimizer estimates and limited fine-grained sampling. Both approaches are expressed as bin-packing problems implemented using iLog [11], a constraint problem solver.

IV. Black-box provisioning

Black-box provisioning profiles the performance and cost of different VM types under varying query input rates using sample executions. Our goal is to capture the input rates each VM can support without violating the QoS of the queries it executes. Based on the profiling results, we identify the min-cost collection of servers to be reserved for each query class in order to complete the workload within its latency bounds.

Problem formulation. In this approach, each virtual server configuration is characterized by a set of *performance vectors*, one for each pair of query class and input rate for that class. We presently evaluate performance vectors once per instance type and deal with predicate uncertainty by running many examples of each query class. Hence, for each combination (i, j, k), where i, $1 \leq i \leq z$ is the instance type, j, $1 \leq j \leq q$ is the query class and k, $0 \leq k \leq m$ is a sample input rate, we create a performance vector $p_{ij}^k = [t_{ij}^k, c_{ij}^k, l_{ij}^k]$, where

1) t_{ij}^k is the expected input rate of the combination (i, j, k), i.e., how many queries of class j it can accept within the reservation period.
2) c_{ij}^k is the cost per query, i.e., $c_{ij}^k = c/t_{ij}^k$, where c is the total operational cost for achieving this input rate.
3) l_{ij}^k is the cumulative distribution function of the query latency for the combination (i, j, k). Also $l_{ij}^k(a)$ returns the $a\%$-percentile latency.

To eliminate any interference among different query classes our provisioning approach does not mix queries from different classes on the same servers. Also, we assume that the workload we are provisioning for consists of n_j queries from each class j that need to be completed within the reservation period. Finally, all queries in a given class j have the same latency expectation, QoS_j.

The decision variable of our problem formulation is $x_{i,j}^k$ and equals to the number of virtual servers of type i and the input rate k they can receive in order to complete all n_j queries of that class within their expectation QoS_j:

$$\min \sum_{i=1}^{z} \sum_{j=1}^{q} \sum_{k=1}^{m} x_{i,j}^k \times c_{i,j}^k \times t_{i,j}^k \quad \text{s.t.} \tag{1}$$

$$\sum_{i=1}^{z} \sum_{k=1}^{m} x_{i,j}^k \times t_{i,j}^k \geq n_j \tag{2}$$

$$\sum_{i=1}^{z} \sum_{k=1}^{m} x_{i,j}^k \times l_{i,j}^k(100) \leq QoS_j \tag{3}$$

The objective function 1 minimizes the total operational cost. Note, that the cost per query might be different for different combinations of VM instance types and input rates. The constraint 2 guarantees that we have reserved enough servers and assigned them incoming rates that allows us to execute all queries of the class j. Constraint 3 ensures that we assign input rates to each server such that all its assigned queries can be executed within their latency constraints.

Input Rates. The black-box approach allows a user to estimate the latency and cost for a given input rate. There is a trade off to be harnessed here between query input rate and latency (QoS). As the input rate goes up, the cost per query goes down as the cost of the reservation period is amortized over more queries. On the other hand, high query rates increase contention leading to higher latency for each individual query.

We demonstrate this tradeoff by profiling a series of performance vectors on an Amazon EC2 instance (m1.large). We varied the time between for each query class from no concurrency (i.e., the length of time for one query to run) to the threshold level (near overload) in declining increments of 10% from the case of no concurrency. These varying query input rates correspond to different QoS (latency) options. Figure 2 shows the trade-off between cost per query and latency bounds.

We observe that relaxing latency requirements does indeed drop monetary costs in most of our queries. In the cases of Q3-Q5, this holds true. Q1 and Q2 are less amenable to having multiple meaningful performance vectors for this instance. Q2 is "lighter" in that it is more likely to be fully cached and less likely to be interrupted, thus for the vast majority of input rates it requires the same latency and monetary cost. Q1 is heavier and naturally overloads the system thus it allows almost no changes in input rate. Thus this trade off of input rate for cost and latency is most interesting in the case of moderate-weight queries that stress the system in distinct ways, allowing them to potentially share resources.

Input Rate	Latency	Instance (Qty)	Cost / Query	Cost / Hour
10	268.65	m1.small (2)	$2.42406	$24.24
14	318.62	c1.medium (1)	$2.42851	$34.00
12	293.22	m1.large (4)	$2.37784	$28.53

TABLE I
INSTANCE SELECTIONS FOR QUERY 4

Illustrative provisioning example. Consider a scenario in which we have a workload specification that calls for 81 queries/hour of Q4 with a latency bound of 333 seconds (this corresponds to 125% of the best latency option available). Drawing on the experiment set used to create Figure 2, we have the options found in Table I. A greedy solution would select 7 instances of m1.large because it has the least expensive cost per query. However our solver selects 4 m1.large instances, 1 c1.medium instance (at 13 queries per hour) and 2 m1.smalls. This provisions for exactly 81 queries as well as saves on reservation costs with the slower instances, while still meeting QoS requirements.

Online query routing. The solution to the above problem specifies the types and number of cloud servers we should reserve for each query class as well as the maximum incoming query rate these servers can handle without violating the QoS goals. At runtime, incoming queries should be routed to the machines executing their query types and in a rate that does not violate their acceptable input rate. This routing problem can also be expressed as a constraint program, which we omit due to space limitations.

V. WHITE-BOX PROVISIONING

The white-box approach solves the resource provisioning problem by estimating the availability and consumption of individual physical resources. First, it quantifies the available resources by profiling the available VM configurations. Next, it evaluates the resource requirements of the target workload using query-specific statistics from the database optimizer paired with a profiling-based scaling factor. Vector representations of both the resource availability and query requirements are evaluated once for an entire workload and then fed to a constraint solver, which then recommends a set of VMs by solving a multi-dimensional bin-packing problem.

Cloud profiling. We assume that the cloud provider offers z different VM configurations, each characterized by d virtual resources. We will use a *resource vector* $r_i = [r_t^0, r_t^1, .., r_t^d]$, $1 \leq t \leq z$ to represent configuration of the computing instance i of type t. Without loss of generality, we assume that $d = 3$ and this vector includes (1) the I/O operations/sec, (2) CPU cycles/sec and (3) the effective memory available in the server.

We estimate I/Os per second by benchmarking Amazon EBS, through a mixed workload of reads, writes and copies

with `unixbench`. CPU and memory availibility are derived from Amazon's specifications.

Query profiling. For each query of a representative workload set $Q = \{Q_1, ..., Q_n\}$, we construct a d-dimensional *work vector* that represents the query's consumption of I/O bandwidth, CPU cycles and memory. We obtain these statistics through a "what-if" interface we built that collects optimizer estimates under different resource configurations. We cope with variety within a query class by experimenting with several (5-10 in our experiments) examples per class.

Most database optimizers generate query plan costs as estimated I/Os and CPU operations, using memory as a constrained resource. For example, in PostgreSQL the `effective_cache_size` parameter is a hint to the optimizer regarding the maximum memory available for that query. Hence, different memory values could lead to varying query plans which impact resource consumption. For example, large memory values will drive the optimizer to pick plans that reduce the number of I/O operations (e.g., opt for hash-joins).

In our PostgreSQL-based implementation, we experimented with different values for the `effective_cache_size parameter`. Our goal was to identify a set of distinct memory allocations that produce different query plans and hence different work vectors, i.e., total resource requirements. We then solve the bin-packing problem by picking only one of these work vectors for each query and assigning it to a virtual machine. We use an "no-execution" query planning mode (PostgresSQL's `EXPLAIN` command) to obtain these hypothetical statistics without executing the query. To obtain the set of work vectors for a given query, we sample the five (5) distinct memory configurations available on AWS. For each sample, we obtain the query's I/O and CPU operation estimates from the optimizer and create a new work vector.

Finally, work vectors and resource vectors need to be compatible in that they must be expressed with the same metrics. Since, I/O (and CPU) resource availability is expressed with rate-based metrics, we use the query-specific QoS targets as our normalizing factor to transform total "total" number of I/O (and CPU) operations to rate-based values. The QoS value represents the maximum tolerable query response time for a query. Specifically, we normalize the optimizer's estimates by the latency expectation for its corresponding query. This gives us the minimum resource usage rate needed to ensure that the query finishes within the specified QoS bound.

Problem formulation. We assume k different work vectors for k different memory values for each of the n queries in our target workload, and $m = n \times z$ available instances, one for each VM type and query statement. This is the maximum number of possible configurations.

Moreover, each query belongs to one of q different query classes and we set $y_{i,j}$ to be equal to 1 if query i belongs to class j. We define $z_{j,u}$ to be equal to 1 if machine j is assigned a query of class u, $z_{j,u} = \max_{1 \le i \le n, 1 \le t \le k} \{a^t_{i,j} \times y_{i,u}\}$.

Finally, we assume each computing instance is rentable for C_t dollars per reservation period and there are per-request charges s_l for each resource type l, $1 \le i \le d$. We use $a^t_{i,j}$ as our decision variable that equals to 1 if the work vector w_{ij} is assigned to the instance t, $1 \le t \le m$. Our constraint programming problem is:

$$\min \sum_{t=1}^{m} (\max_{1 \le i \le n, 1 \le j \le k} a^t_{i,j}) \times C_t \quad + \quad \sum_{l=1}^{d} (\sum_{t=1}^{m} \sum_{j=1}^{k} \sum_{i=1}^{n} a^t_{i,j} \times w^l_{i,j}) \times s_l \quad \text{s.t.} \tag{4}$$

$$\sum_{t=1}^{m} \sum_{j=1}^{k} a^t_{i,j} = 1, \qquad \forall i, i \in \{1, ..., n\} \tag{5}$$

$$\sum_{j=1}^{k} \sum_{i=1}^{n} a^t_{i,j} \times w^l_{i,j} < r^l_t, \quad \forall t, t \in \{1, ...m\}, \forall l, l \in \{1, ...d\} \tag{6}$$

$$a^t_{i,j} \in \{0, 1\}, \quad \forall i, i \in \{1, ...n\}, \forall j, j \in \{1, ...k\}, \forall t, t \in \{1, ..., m\} \tag{7}$$

$$\sum_{u=1}^{q} z_{j,u} = 1, \qquad \forall j \in \{0, .., m\} \tag{8}$$

The objective (Eq. 4) aims to minimize the total operational cost (the cost of renting the virtual machines for a single reservation period as well as any per-request costs). Trying to minimize both charges will drive our solver towards solutions that minimize the number of machines we use, reserve the cheapest machines when possible, and also choose a set of work vectors that incur the lowest possible discrete cost per resource (e.g., have the least possible I/O operations). Constraint 5 guarantees that our solution uses exactly one work vector from each query in the given workload. Constraint 6 ensures that each reserved machine has enough capacity to cover the resource requirements of its assigned work vectors. Given the solution to the above problem it is straightfoward to identify the number and types of virtual machines we need to reserve is. Finally, constraint 8 guarantees that each server will be assigned queries of the same class.

Online query routing. At runtime, user queries can be executed on any of the reserved machines, however, certain query assignments could yield lower operational costs. For instance, a machine with higher available memory could execute query plans with fewer I/O operations. We assign incoming queries to the reserved machines in two steps. First, we construct the work vectors of the new queries. Second, we solve an incremental bin-packing problem that is cognizant to the currently executing queries and the residual resource availability. We omit the details due to space constraints.

Calibrating Estimations. Our white-box approach relies on the accuracy of its work vectors. To assess the accuracy of the estimations, we ran several TPC-H queries in isolation and quantified how many I/O operations and CPU cycles were consumed. By running the queries alone with a cold cache, we have a conservative estimate of the resources consumed. Thus, this approach gives us the maximum slice of resources that a query class will require. We experimented with five different query examples for each of TPC-H queries Q1-Q5.

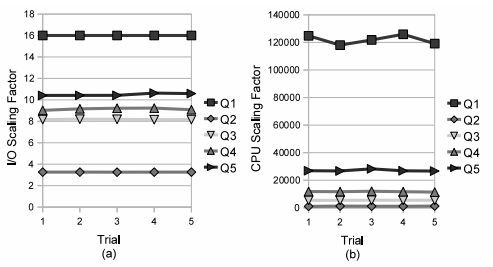

Fig. 3. Scaling factors for (a) I/O and (b) CPU operations.

We use optimizer estimates to guide our work vectors. These estimates are typically provided as abstract units, which need to be translated into "real" resource consumption values. Our experiments demonstrated that a simple, linear scaling factor (ratio of the actual resources consumed and the optimizer's estimate for that dimension) per query class could perform this calibration accurately for the TPC-H queries we considered. In Figure 3 we see the scaling factors for the CPU and I/O estimates for our work vectors. PostgreSQL considers all of its cost parameters in terms of time (their unit is comparable to the time it takes to do one sequential read), so we must scale it into resources used by converting I/O time into actual I/Os and CPU time into cycles to normalize for clock speed.

The scaling factors for the I/O operations vary for different query classes. The variation between classes are due to different degrees of errors the optimizer makes in its estimations, which are largely based on the complexity of the query plans and skewed data distributions. Furthermore, scale factors may also vary considerably between instances based on cache size. The optimizer has only a very coarse knowledge of the buffer pool usage, which leads to further errors. The scaling factors for CPU usage normalization have similar properties. In absolute values, they are much larger because we convert from time units to cycles to be able to make EC2 instances with varying compute unit power comparable.

While the scaling factors varied between queries, they were relatively stable across queries in the same class. This observation provides a motivation for not executing queries from different classes on the same machine for predictability reasons. Each query does not vary much in its scaling factor despite their actual cost differing based on predicate ranges, index selection and other path decisions. Our standard deviation for scaling factors vary between 0-7% of the average for each query class and similar results were obtained from different EC2 instances.

Instance Selection. Our provisioning aims to select the "cheapest" VMs that satisfy the workload's QoS goals. Our goal is to determine the cost/performance trade-off and demonstrate that varying cloud server configurations yield varying QoS levels and operating costs. To study the utility of our approach we needed to determine whether the bin packer had many competing options. Otherwise, if there is one clear VM option a greedy solution would be sufficient.

To address the above, we experimented with a variety of workload specifications for Q2. Specifically, we executed ten different instances of the class Q2 on an m1.xlarge EC2 instance and varied the query input rates and latency requirements. Workloads with faster input rates had more relaxed latency requirements. In contrast workloads with slower input rates specified much tighter latency requirements. We calculated the cost per query as our reservation cost (divided by the number of queries executed in the reservation period) plus the average I/Os used per query multiplied by its EBS rate. We selected the best instance based on the least expensive cost per query that met our QoS requirements.

Input Rate (/ hour)	Latency Req	Best Inst	Cost / Query	Cost / Hour
720	30	c1.xlarge	$0.00344	$2.47590
360	30	m1.large	$0.00120	$0.43109
240	20	m1.large	$0.00145	$0.34912
180	20	m1.large	$0.00192	$0.34508
144	15	c1.xlarge	$0.00474	$0.68300
120	15	c1.xlarge	$0.00569	$0.68260
103	12	c1.xlarge	$0.00663	$0.68257

TABLE II
INSTANCE SELECTION FOR VARYING WORKLOADS.

Our findings are in Table II. In this experiment our cost per query changed as the input rate and latency requirements. For low input rates Q2 starts out favouring a compute-intensive instance (c1.xlarge) as the CPU is the resource with the most contention. As we move on to a more moderate input the preference switches to a memory-intensive instance (m1.large) to minimize expensive I/Os used in page swapping. In the final trials the tighter QoS causes a CPU bottleneck, so we revert to c1.xlarge. These trials clearly demonstrate that the instance to be selected will vary based on workload specifications.

VI. Related Work

Recently, there has been enormous interest in the area of cloud-based data management. Several groups (e.g., [2], [3]) discussed the benefits, drawbacks and challenges from moving data management applications and tools into IaaS-based machines, although they do not address the resource provisioning problem nor the cost/performance trade-offs that arise from the elastic model of cloud infrastructures. However, the need for including monetary cost as a database optimization objective was acknowledged [8].

While a utility-driven solution for adaptively modifying workload allocation within a cloud was presented in [14], this work has not yet been extended to support monetary cost and was not focused on cloud databases. In [5] and [12], the authors focus on maintaining various levels of ACID properties for databases on the cloud. While they do address financial cost, they do not broach the issue of performance while minimizing cost. In [6], the authors analyzed the changes in price versus latency for different degrees of parallelism for scientific workloads by using simulations of Amazon's AWS. In contrast, we are working with data warehousing workloads and using real deployments for our source data and evaluations.

Our framework is related to techniques for configuration of virtual machines [16] and automatic performance modeling of virtualized applications [15]. There has also been a substantial amount of work on the problem of tuning database systems [19] and adapting databases to their use of various

Query	1	2	3	4	5
Std. Dev	11.9	1.09	18.92	19.07	10.74
Avg. Latency	619.42	20.23	268.62	300.38	255.09

TABLE III

STATISTICS FROM WORKLOAD AT VARIOUS TIMES OF DAY.

computing resources [13], [17]. However, in this work we provision resources aiming to improve the monetary cost under certain performance constraints. To the best of our knowledge, such a tuning objective has yet to be studied. Our black-box approach and optimizer scaling can borrow from the more sophisticated learning-based predictive modelling [9].

Resource provisioning has been addressed from a profiling perspective [18] and model-based approach [7] in other areas of systems research. These approaches were not specifically geared toward database systems, nor did they have a flexible enough approach for dealing with monetary cost. We address both of these topics in our research.

VII. Open Challenges and Ongoing Work

Here we briefly summarize several open issues and our ongoing efforts to address them.

Mixed Workloads. Accurately modelling the behavior of mixed workload execution is a key challenge. We plan to explore systematic profiling to characterize the sensitivity of query classes to mixed execution, along with predictive techniques to do this with acceptable accuracy and cost in a manner similar to [4].

Probabilistic Modeling of Resources and Queries. There is often a lot of variation and unpredictability in the workload, resource availability, and performance, as well as the accuracy of the estimates. A simple, discrete modeling of such a complex environment is unrealistic and fundamentally lead to solutions that are overly conservative. We are currently investigating probabilistic modeling of our resource and work vectors to alleviate these problems.

To quantify some of this uncertainty in cloud resources, we ran TPC-H queries Q1-Q5 every 6 hours for five days on an m1.large EC2 instance with fixed predicates and plotted execution latencies. Table III shows that, in this case, the standard deviation of latencies was small enough to facilitate effective provisioning but not too small to be ignored.

Expressing Domain-Specific Policies. A further utility of our generic approach is in expressing domain-specific workload allocation policies. Such policies could be either driven by system model constraints or heuristics that could potentially improve the performance of the system. For example, if a master-slave architecture is used then update queries need to be forwarded to the master machines and read operations to the slave machines. Or, we might want to exploit cache locality by executing queries accessing the same tables in the same machine. Such scenarios can be readily expressed within our framework through additional constraints.

Fine-grained Resource Allocation. Our white-box solution assumes that specific slices of CPU and I/O can be allocated for each query. Most engines do not provide such a fine-grained control over how resources are allocated internally. In those cases, we can still achieve a workable solution if we know what allocation policies an engine uses. With PostgresSQL, for example, a fair scheduling assumption that gives queries a uniform share of the available resources seems to be a realistic one. Our generic constraint programming-based framework allows us to integrate such knowledge into the solutions easily.

Gray-Box Formulations. The black-box approach requires extensive experimentation, whereas the white-box approach requires good optimizer estimates and some control over (or, at the very least, the knowledge of) how resources are allocated within the database engine. A hybrid approach that combines the respective strengths of these two points in the solution space is an area which we are actively pursuing.

VIII. Acknowledgements

This work has been partially supported by the National Science Foundation under grant No. IIS-0916691 and IIS-0905553.

References

[1] Amazon Web Services, http://aws.amazon.com/.
[2] D. J. Abadi. Data management in the cloud: Limitations and opportunities. *IEEE Bulletin on Data Engineering*, 32(1), 2009.
[3] A. Aboulnaga, K. Salem, A. A. Soror, and U. F. Minhas. Deploying database appliance in the cloud. *IEEE Bulletin on Data Engineering*, 32(1), 2009.
[4] M. Ahmad, A. Aboulnaga, S. Babu, and K. Munagala. Modeling and exploiting query interactions in database systems. In *CIKM '08: Proceeding of the 17th ACM conference on Information and knowledge management*, pages 183–192, New York, NY, USA, 2008. ACM.
[5] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on s3. In *SIGMOD*, 2008.
[6] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The cost of doing science on the cloud: the montage example. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.
[7] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat. Model-based resource provisioning in a web service utility. In *USITS*, 2003.
[8] D. Florescu and D. Kossman. Rethinking cost and performance of database systems. In *ACM SIGMOD Record*, March 2009.
[9] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *ICDE*, 2009.
[10] GoGrid.com. http://gogrid.com/.
[11] IBM. ILOG CPLEX, http://www.ilog.com/products/cplex/.
[12] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: Pay only when it matters. In *VLDB*, 2009.
[13] D. Narayanan, E. Thereska, and A. Ailamaki. Continuous resource monitoring for self-predicting DBMS. In *MASCOTS*, 2005.
[14] N. W. Paton, M. A. T. de Arago, K. Lee, A. A. A. Fernandes, and R. Sakellariou. Optimizing Utility in Cloud Computing through Autonomic Workload Execution. *IEEE Bulletin on Data Engineering*, 32(1), 2009.
[15] P. Shivam, A. Demberel, P. Gunda, D. Irwin, L. Grit, A. Yumerefendi, S. Babu, and J. Chase. Automated and On-Demand Provisioning of Virtual Machines for Database Applications. In *SIGMOD*, 2007.
[16] A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *SIGMOD*, 2008.
[17] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in DB2. In *VLDB*, 2006.
[18] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting plat forms. In *OSDI*, 2002.
[19] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. In *VLDB*, 2002.