Tossing NoSQL–Databases out to Public Clouds

Alexandros Antoniadis, Yannis Gerbesiotis, Mema Roussopoulos and Alex Delis
University of Athens, Athens, Greece
Email: {a.antoniadis, j.gerbesiotis, mema, ad}@di.uoa.gr

Abstract—Cloud-Service Providers (CSPs) can now handle heavy workloads by occasionally renting resources from public clouds. The capabilities and respective lease prices of such infrastructure may significantly vary over time. In this environment, two distinct types of SLAs have to work in tandem: a) the SLA furnished by the private cloud to the end user of the application (or database), and b) the SLA offered by the public cloud to the application through its host private cloud. This dual and continuously evolving relationship inherently complicates the computation of the operation of cloud applications. In this paper, we present a cost-aware resource provisioning algorithm for NoSQL-databases that aims to meet Quality of Service (QoS) requirements while minimizing the total cost incurred by its deployment on multiple cloud tiers. Our method is based on look-ahead optimization and takes into account the costs incurred by potential database transitions to new configurations in a heterogeneous multi-cloud environment. Experimentation with a prototype shows that our approach reflects the total cost of a cloud application more accurately than the conventional technique of minimizing SLA violations. More importantly, it avoids thrashing of resources.

Keywords—resource provisioning; NoSQL-databases; look-ahead optimization

I. INTRODUCTION

Cloud-Service Providers (CSPs) dynamically offer computational and storage resources so that users can experience timely execution of their applications regardless of the load and queued jobs the infrastructure has to handle [1]. CSPs have the freedom to calibrate both type and number of allotted resources at different points in time so that incoming workloads are successfully handled. In such settings, QoS guarantees regarding performance aspects such as response time, throughput, and service availability can be provided to both user applications and launched databases through the use of SLAs. When SLA violations occur, monetary penalties are accrued for the CSP directly affecting not only its revenue but more importantly, its reputation [2].

Untimely provisioning by a CSP of its own internal (or private) resources can lead to depressed leasing costs that ultimately prevent application QoS-requirements from being met. Resources needed by an application might change either periodically (i.e., high peak hours or days) or irregularly (i.e., flash crowds that cause sudden, significant depletion of resources). A CSP could address internal resource shortages by soliciting additional resources that are available just-in-time from external or public CSPs. Dynamic allocation/deallocation of cloud resources might help, but frequent workload changes may lead to deployment thrashing as overheads incurred by the additions/removals of resources may outweigh any short-term benefits gained. To complicate matters further, pricing for leasing equivalent resources from public CSPs continuously fluctuates. The latter has to be taken into consideration to identify a resource allocation with minimum cost. It becomes evident that resource allocation is not a straightforward task and so it has recently attracted considerable attention [3]–[5].

In this paper, we investigate the problem of provisioning a popular class of cloud applications collectively known as NoSQL-databases [6], [7]. Their key characteristic is that they can scale their performance as they offer horizontal partitioning of data in a shared-nothing fashion through sharding [8].

NoSQL-databases are typically designed to provide availability and fault tolerance by replicating their data multiple times on different nodes across Gbps-interconnected cluster(s). The notion of cluster here is that of a set of network-connected machines possibly having different hardware features. As nodes arrive at or depart from the cluster, (e.g., because of energy concerns), replicas have to be expanded or contracted respectively so that availability remains intact. Such “transitions” however expend computational, storage, and network resources and thus, do not occur instantly. This is a key aspect that one has to consider when it comes to NoSQL-database provisioning and possibly soliciting resources from external or public CSPs.

We present a resource provisioning approach that exploits the pricing models of available resources as well as the costs imposed by potential movements of shards. We aim to minimize the total cost of running a cloud application by using look-ahead optimization for a limited time-window. Fig. 1 depicts the key aspects of our approach. The private CSP delegates the selection of resources needed to run an application to the provisioning algorithm based on look-ahead optimization that oversees the minimization of the total cost. As a result, parts of the database may be “tossed out” to public CSPs to expedite processing. Our approach has two phases: the first phase consists of profiling the application so that a performance model for its execution in a specific sample of cluster configurations is built; a cluster configuration consists of a specific combination of machines that form a cluster. Several executions of the application on different cluster configurations are needed to stress the application and build a respective performance model. The performance model is required to estimate the behavior of the application on future cluster configurations. Although the creation of a performance model is a costly task in terms of time, it is carried out just once. The second phase requires the following pieces of information (Fig. 1): i) the derived performance model, ii) the
application SLAs, iii) the prediction of the upcoming workload, and iv) the available resources from the private and/or public CSPs. Using the above information, the resource provisioning algorithm designates which of the available resources should be either added or dropped so that the cost of operating the private CSP remains at a minimum.

We make the following contributions:

1) We expose key factors that should be considered in provisioning as they affect the private CSP cost either directly or indirectly. We develop a comprehensive cost model to account for all expenses involved including penalties that have to be “paid back” by public CSPs should they violate their own SLAs. We also introduce the transition cost needed to re-host a portion of an application and examine how this affects the total cost through experiments.

2) We address the NoSQL-databases provisioning problem taking into account the perspective of the private CSP hosting database shards. Thus, we focus on minimizing the private CSP’s cost and compare this with the widely-accepted approach of reducing penalties incurred by SLA violations.

3) We introduce a look-ahead optimization-based provisioning approach and investigate its effectiveness in comparison with competing approaches including resource thrashing avoidance [4], [9].

The paper is organized as follows: Section II, outlines the salient factors that affect the provisioning problem. Section III discusses both our profiling and the techniques we use to create a predictive model for the cluster. Section IV presents our look-ahead provisioning algorithm and Section V presents key experimental results. Finally, Sections VI and VII respectively describe related work and concluding remarks.

II. FACTORS IN CLOUD PROVISIONING

We outline key factors that should be considered while provisioning for NoSQL-databases in the private/public CSP context.

A. Opportunistic Use of Public Cloud

When private CSPs rent additional machines from public CSPs to auto-scale NoSQL-databases, they form “clusters” of virtual infrastructures that go beyond what they have available locally. This may transparently offer substantial benefits to users as they see their applications “grow” without necessitating the purchase of new machinery but only the occasional leasing of resources. This leasing highly depends on i) the specification of the machine(s) needed, and ii) the SLA that the public CSP offers for the request. Differences between nodes that belong to public clouds and nodes in a private CSP include the following: 1) public CSP nodes have a rental/lease cost while private nodes have an operational cost that entails both energy and maintenance costs, and 2) a public cloud has to compensate the private cloud in the form of monetary penalty or pay-back anytime an SLA is violated. Imposed penalties on public CSPs indirectly affect the cost calculation that a NoSQL-database host has to pay to a user should SLA violations be certified. Thus, the entire amount of penalty is not exclusively paid out by the private CSP running an application. This is a critical factor that should be taken into account when designing a resource provisioning algorithm.

B. Transitions

Every node addition to or removal from a NoSQL-database does not happen instantly. The time it takes for a new node to become operational in a cluster or an operating node to cease operation may range from a few milliseconds to several minutes or hours. A newly instantiated node might need to deploy software artifacts, edit property files and/or start groups of services. Also, in NoSQL-databases, a node addition means that data will typically be shipped and replicated over the network. Apart from any requisite data transfer, the cluster may need to update its own internal data structures and indices to reflect the new state. The above requisite operations consume resources from both new and old nodes and may entail major overheads in terms of CPU-cycles, memory, disk and occasionally, network bandwidth. If these operations are not carefully considered, they could easily push the cluster into an unstable state.

C. Cluster Heterogeneity

Cloud infrastructure typically exhibits significant heterogeneity in terms of CPU, memory, disk(s) and NICs of cloud infrastructure nodes [2], [10]. This is due to private CSPs incrementally upgrading their internal machinery as well as public CSPs competing against each other and frequently changing their rental offerings to better match client requests [11]. An adaptive cloud-based application that aims to exploit the best out of the available resources should leverage the heterogeneity of cloud machines accordingly. In our work, we handle this problem by profiling NoSQL-databases such as the ELASTICSEARCH [6] under different cluster configurations. We use linear regression and support vector regression to predict performance metrics of future deployment outcomes such
as estimated throughput and expected percentage of operations with latencies that violate the SLA of the application.

III. BUILDING AN APPLICATION PERFORMANCE MODEL

To effectively decide which nodes will be part of a cluster, we want to successfully estimate the behavior of the purported configuration once provisioning takes place. We accomplish this by creating a performance model for the NoSQL-database under deployment, so we use this model to estimate the cost of a newly introduced configuration as SLA violations can be traded off with leasing costs from public CSPs. Predicting the performance of the NoSQL-databases with reasonable accuracy is key to our provisioning method. As queueing-analytic based models cannot deliver viable solutions [12], [13], we resort to an empirical modeling approach. Our approach initially carries out selective stress tests on different cluster configurations for the NoSQL-database at hand [14]. As this process is carried out offline, no penalties are imposed during the normal execution of the NoSQL-database. It then creates a forecasting model to offer configuration suggestions based on data collected. We use linear regression and support vector regression to predict the performance of potential cluster configurations to determine whether tossing NoSQL-databases out to public CSPs is beneficial.

A. Profiling Experiments

We use a modified version of the Yahoo! Cloud Serving Benchmark (YCSB) [15] to profile ELASTICSEARCH v0.20.6, a popular NoSQL-database that uses sharding [8] to distribute horizontal partitions of data to different Virtual Machines (VMs). ELASTICSEARCH v0.20.6 tends to distribute its shards equally to all of nodes that participate in the cluster; to this end, we seek to ascertain how ELASTICSEARCH behaves under variable number of CPU–cores, CPU–frequency, memory, and VMs.

In the standard YCSB edition, a client either creates or joins an existing cluster of nodes. Hence, it is likely that at least a portion of the requested data may reside in a shard located on the client’s VM. This is surely an unusual setting for a cloud setup as in cloud environments, the back-end components handling data are often separate from application clients. ELASTICSEARCH features non-data nodes that can function as load-balancers. In our modified YCSB, a client simply connects to a load-balancer node to access data from all shards dispersed throughout the network. This layout better reflects realistic deployments of NoSQL-databases.

We perform a number of YCSB runs on ELASTICSEARCH with different targeted throughput in clusters with up to 6 nodes and a single load-balancer node while varying the number of CPU–cores, CPU–frequency, memory, and VMs of the cluster. We added 3,000,000 records to the ELASTICSEARCH database and then performed 500,000 GET operations following a uniform distribution based on the record ID. We measured the following:

- throughput of each cluster configuration,
- percentage of operations per unit time that violates SLAs; we term this as DROP: delayed response operations percentage,
- transition cost and delay for data–node addition/removal.

We assume that a request completing in more than 5ms generates an SLA violation. For brevity, we omit showing the profiling experiments for CPU–frequencies. In general, the CPU–frequency profiling results follow similar trends with those of RAM. Below, we outline the key findings from profiling the above NoSQL-database in a private CSP.

- **Throughput**: Fig. 2a shows that as VMs are added, throughput increases almost linearly. The same behavior is observed for CPU–cores as more GET operations can be handled. Similar trends are depicted in Fig. 2b, where the size of RAM varies. We find that throughput is affected less by RAM and CPU–frequency as GET operations scale out better (i.e., adding more VMs) than scale up (i.e., increasing memory and CPU–frequency).

- **DROP**: Results for requests missing their SLAs are not as clear cut as those of throughput. Fig. 3a reveals that the resulting DROP maintains high margins between the average value and the maximum and minimum values attained as we increase the number of VMs. Similar observations hold for DROP rates while varying CPU–cores and RAM in Figs. 3b and 3c; they demonstrate behavior with no clear trends. Thus, the above three profiling viewgraphs cannot lead to any strong conclusions regarding DROP prediction. Consequently, we resort to machine learning techniques to more effectively forecast DROP values.

- **Transition Cost and Delay**: We have performed experiments where we add or remove data–nodes and monitor how the throughput of the cluster is affected. In these profiling experiments, we ascertained that the transition cost (i.e., transporting shards) is almost independent of the configuration of the nodes that participate in the cluster. Although bandwidth linearly affects costs and delays, in an environment where multi-Gbps networks connect private and public CSPs, this factor becomes an invariant for modeling purposes.

B. Forecasting Models

Our forecasting model takes as input a set of VMs to be possibly incorporated into the operational cluster along with a number of parameters that include: i) public CSPs from which to lease the VMs, ii) CPU–cores, iii) CPU–frequency, iv) size of RAM, v) number of VM nodes. The outcome of the forecasting model states how the re-aligned cluster would perform should the additional VMs from the public CSPs be included as part of the cluster. The output of the model consists of the following anticipated rates and/or values: I) throughput rate, II) DROP rate, as well as III) transition cost, duration, and delay. Below, we discuss how we deliver these three rates and costs.

The outcome of our black-box profiling yields selected measurements for specific coordinate values in a multidimensional...
(a) Throughput: Operations per second over VMs # and CPU–cores

(b) Throughput: Operations per second over RAM

Fig. 2: ELASTICSEARCH throughput

(a) DROP over VMs #

(b) DROP over CPU–cores

(c) DROP over RAM

Fig. 3: Percentage of operations missing their SLA (DROP) in ELASTICSEARCH
space. If we knew every possible value in this space, we would be able to derive the best solution for a given provisioning. However, this is infeasible, so we use approximate estimation methods to produce the output rates/values of the model. In particular, we use predictive techniques [16] to estimate expected rates for throughput and DROP. These techniques need an adequate size of training data to be well calibrated. Moreover, the training set has to consist of a representative sample of cluster configurations to both accurately predict the future and remove outliers. We experimented using the RapidMiner [17], a software platform for machine learning, to ascertain pros and cons of various predictive techniques and we have identified the following options (cases I and II below) to respectively estimate throughput and DROP rates:

I) Linear Regression for Throughput: Fig. 2a clearly shows that the cluster throughput increases linearly with the number of VMs and/or CPU–cores. Consequently, using linear regression to estimate throughput rates for cluster configurations that have not been evaluated in the stress-test profiling phase is the intuitive choice. In contrast, the addition of RAM in the cluster leads to less discernible gains for throughput (Fig. 2b). A similar trend to that of RAM occurs with CPU–frequency as well. While using linear regression for throughput estimation, we place less importance on the RAM and CPU–frequency values than the number of VMs and CPU–cores used by using appropriate weights; the latter are computed during the fitting process.

II) Support Vector Regression (SVR) for DROP Rate: Fig. 3 collectively reveals that although the average DROP rate decreases as the values of input variables increase, the minimum-to-maximum range for resulting DROP values remains large. There are undoubtedly complex relationships between the five input variables and the expected DROP rate that are impossible to capture using linear estimation techniques. The presence of multidimensional variables along with their complex relationships makes the SVR approach suitable for our case as it can more effectively predict the DROP rate.

SVR maps data from their original feature space into a higher–dimension feature space and then computes an optimal regression function in this new feature space [18]. This data transformation is carried out through the mapping: \( v \rightarrow \varphi(v) \), where \( v = (v_1, v_2, \ldots, v_n) \) is a vector of independent variables; in our case, \( v \) represents the \( n \) values of our input variables \(^1\) making up a single data point. The mapping is assisted by kernels that essentially bypass the explicit use of \( \varphi(\cdot) \) to transform data to the new feature space. Kernels are realized as the dot product of two vectors \( i \) and \( j \) in the feature space as follows: \( k(v_i, v_j) = \varphi(v_i) \cdot \varphi(v_j) \). Among popular non-linear kernels, we use the Gaussian Radial Basis Function (RBF) as the DROP rate depicts non-linear behavior and RBF proves to be the most accurate. The RBF kernel is: \( k(x_i, x_j) = \varphi(x_i) \cdot \varphi(x_j) = e^{-\gamma ||x_i - x_j||^2} \), where \( \gamma \) is an adjustable positive variable.

III) Transition Cost and Delay: Using additional VM(s) from possibly different public CSPs involves a delay that is required to ship a shard to the designated VM(s). This transition can be expressed as a function over time as follows:

\[
\text{transition}(t) = \begin{cases} 
0 & t \in [0, \text{delay}] \\
\text{tr\_overhead} & t \in (\text{delay}, T]
\end{cases}
\]

where \( \text{delay} = D_{\text{startup}}+D_{\text{shutdown}} \),
\( T = \text{delay} + \text{tr\_duration} \),
\( \text{tr\_duration} = \text{duration\_per\_shard} \cdot \text{moved\_shards} \),
\( \text{tr\_overhead} = \text{overhead\_per\_shard} \cdot \text{moved\_shards} \)
and \( \text{moved\_shards} = \text{max} \)
\( \text{added\_nodes} \cdot \left( \frac{\text{total\_shards}}{\text{new\_cc\_nodes}} \right) \),
\( \text{removed\_nodes} \cdot \left( \frac{\text{total\_shards}}{\text{cc\_nodes}} \right) \).

Here, \( D_{\text{startup}}+D_{\text{shutdown}} \) represents the fixed time that a VM takes to either start up or shut down, \( \text{moved\_shards} \) is the number of shards to be moved when the cluster configuration changes, \( \text{added\_nodes} \) and \( \text{removed\_nodes} \) are the number of the nodes added to or removed from the cluster configuration respectively, \( \text{new\_cc\_nodes} \) and \( \text{cc\_nodes} \) are the number of the nodes in the new and current cluster configurations respectively, \( \text{total\_shards} \) is the number of shards involved in the NoSQL-database, \( \text{duration\_per\_shard} \) is the overhead, in seconds, that each moved shard adds and \( \text{overhead\_per\_shard} \) is the overhead in operations per second that each moved shard generates. Table I shows observed average values of representative factors involved in the estimation of Transition Cost and Delay. Multiple experiments yield invariable values indicating constant overhead behavior.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D_{\text{startup}} )</td>
<td>3 secs</td>
</tr>
<tr>
<td>( D_{\text{shutdown}} )</td>
<td>2 secs</td>
</tr>
<tr>
<td>( \text{duration_per_shard} )</td>
<td>4 secs</td>
</tr>
<tr>
<td>( \text{overhead_per_shard} )</td>
<td>1000 Mbps</td>
</tr>
<tr>
<td>( \text{total_shards} )</td>
<td>10</td>
</tr>
</tbody>
</table>

IV. LOOK-AHEAD OPTIMIZATION FOR CSPs

Our main objective is to identify the least expensive combination of nodes that collectively satisfies the constraints imposed by the cloud applications(s). These constraints can be either strong (i.e., calling for SLA penalty minimization) or weak (i.e., seeking to lower the CSP expenses). Either way, the selection of VMs and the identification of a “cluster” to be used highly depend on the current configuration on which the application runs as well as its anticipated workload characteristics. [10] showed that service provisioning is NP-hard and suggested heuristics to prune the solution space by

\(^1\)In particular, five values corresponding to (i) public CSPs used, (ii) CPU–cores, (iii) CPU–frequency, (iv) size of RAM and (v) number of VMs.
limiting either the depth of the ensued search tree or the time period within which a viable solution is sought.

We employ *Look-Ahead Optimization (LAO)* to identify a (sub)optimal selection of a new cluster configuration by examining all possible paths that are feasible at a specific point in time. Our approach uses the current state of affairs within the cluster and seeks to optimize future states. As in [5], we assume a relatively accurate predictor for workload characterization to predict the outcome of a future cluster configuration.

### A. Receding Horizon Control (RHC)

is a LAO–method that iteratively solves an optimization problem for a fixed time interval while taking into account current and future constraints; it has been successfully used for resource provisioning [4]. *RHC* functions in a recurrent fashion as follows:

**S1)** At time \( k \), find an optimal solution for the specific and fixed–period \([k, k + T]\) while considering current allocations and forthcoming constraints.

**S2)** Apply only the first element of the above optimal sequence.

**S3)** Shift time \( t \) to \( k + 1 \) and repeat the process for the interval \([k + 1, \ldots, k + T + 1]\).

Should there be no (other) external factors that affect the cost computation of the solution sought in step \( S1 \) above, the *RHC* finds the optimal solution for the given time-window \( T \).

We assume the following:

- **A1)** \( J \) represents the sum of the current\(^2\) operational/leasing cost of the VM resources placed in a cluster from both private and public CSPs, the costs of incurred SLA violations and public CSP pay-backs, and the imposed transition cost for a unit of time,
- **A2)** \( x_t \) represents the state of the cluster in terms of the set of alloted resources at time \( t \),
- **A3)** \( u_t \) entails all feasible transitions to reach a new cluster configuration at time \( t \); this set of transitions involves additions or removals of VMs,
- **A4)** \( \{x_i\} \) is the sequence of all states generated in the period \( k, \ldots, t \),
- **A5)** \( \{u_i\} \) is the sequence of all transitions that have taken place within period \( k, \ldots, t \),
- **A6)** \( x_{i+1} = f(x_i, u_i) \) for \( i = k, \ldots, k + T \), where \( f \) is the function that maps a state \( x_t \) to the next \( x_{t+1} \) according to \( u_t \) input choices available at time \( t \),
- **A7)** \( \text{cost}(\{x_i\}, \{u_i\}) = \sum_{i=k}^{i=k+T} J(x_i, u_i) \) represents the cumulative cost incurred while following the corresponding \( \{u_i\} \) sequence and \( J \) is the cost function defined above in \( A1 \).

In step \( S1 \) of the *RHC*, we identify the optimal solution as the one that provides as \( \text{cost}_{\text{opt}} = \min \text{cost}(\{x_i\}, \{u_i\}) \). The solution of the above optimization problem leads to a sequence of suggested cluster configurations \( \{x_k, \ldots, x_{k+T}\} \) and a respective sequence of transitions \( \{u_k, \ldots, u_{k+T}\} \) that eventually take place. The sequence \( \{x_k, \ldots, x_{k+T}\} \) corresponds to a path having the minimum cumulative cost in the time–window elapsed between \( k, \ldots, k + T \).

### B. Selecting the Time-Window Period

The time-window is a fundamental *RHC* parameter as it designates the depth in which a solution is to be searched and presents a number of trade-offs. On the one hand, a short window might miss a number of good long-term changes if it cannot capture significant future workload changes. On the other hand, a long time-window affects the execution time of the algorithm as it may introduce exponential complexity. A viable choice for time-window length should be able to capture at least a few complete transitions in the make up of a cluster as well as pertinent overheads. Any benefits in the operation of a re-aligned cluster will be reaped after the transition eventuates. Hence, it is also imperative that the time-window be a function of the average duration of the transitions.

### C. Resource Provisioning Algorithm

Algorithm 1 recursively determines the cost as well as the entire sequence of cluster configurations generated within the time-window \([\text{start\_time}, \text{end\_time}]\) that imposes minimum cost for the private CSP (\( \text{best\_configs} \)). Starting from the initial cluster configuration \( \text{cc} \), the algorithm examines all possible configurations that can be reached while trying to identify the next configuration possibly involving VMs from public CSPs as well. The invocation of \( \text{possible\_cluster\_configs}(\text{cc}) \) produces feasible configurations by taking into account the replication factor of the NoSQL-database. The replication factor designates the number of redundant copies of shards, and so, it limits the number of VMs that can be removed from a cluster during a single transition.

**Algorithm 1 Provisioning Best-Plan**

```plaintext
procedure \( \text{BEST\_PLAN}(\text{cc}, \text{start\_time}, \text{end\_time}, \text{best\_cost}, \text{best\_config}) \)
for all \( \text{cl} \) in \( \text{possible\_cluster\_configs}(\text{cc}) \) do
    \( \text{tr\_delay}, \text{tr\_duration}, \text{tr\_overhead} = \text{transition}(\text{cc}, \text{cl}) \)
    \( \text{time} = \text{start\_time} \)
    if \( \text{tr\_delay} + \text{tr\_duration} + \text{tr\_overhead} > \text{end\_time} \) then
        \( \text{cost} = 0 \)
        \( \text{tr\_delay} = 0 \)
        \( \text{tr\_duration} = \text{end\_time} - \text{start\_time} \)
    end if
    \( \text{cost} = \text{PARTIAL\_COST}(\text{cc}, \text{cl}, \text{tr\_duration}, \text{tr\_delay}, \text{tr\_overhead}, \text{start\_time}) \)
    \( \text{time} += \text{tr\_delay} + \text{tr\_duration} \)
    \( \text{configs} = \{\text{cl}\} \)
    if \text{time} < \text{end\_time} then
        \( \text{p\_cost, configs} = \text{BEST\_PLAN}(\text{cc}, \text{cl}, \text{time}, \text{end\_time}, \text{best\_cost}, \text{best\_configs}) \)
        \( \text{cost} += \text{p\_cost} \)
    end if
    if \text{cost} < \text{best\_cost} then
        \( \text{best\_cost} = \text{cost} \)
        \( \text{best\_configs} = \text{cl} + \text{configs} \)
    end if
end for
return (\text{best\_cost}, \text{best\_configs})
end procedure
```

A possible change in cluster configuration implies transition costs for resource re-alignment that may require non-negligible operations and takes a duration interval to unfold. Moreover, the transition may have a latency, termed \( \text{delay} \), before it actually completes. \( \text{transition()} \) estimates for transition delay, duration and overhead based on the suggested performance model of Section III. These three values, along with current and a feasible new cluster configuration, are furnished to Algorithm 2 (\( \text{PARTIAL\_COST} \)) to assess the cost of a proposed transition; the latter is essentially the factor \( J \) defined in \( A1 \) above. Subsequently, Algorithm 1 shifts the \text{start\_time} of the
time-window by as much as the time required to complete the suggested transition. The algorithm then moves to compute the rest of the optimal cluster configuration sequence in a recursive manner always using the first element of the remaining sequence as the pivot for its exploration. In this regard, the recursive calls along with the loop over the set produced by POSSIBLE_CLUSTER_CONFIGS(), build a tree with all feasible configuration sequences within the sought time-window. While this tree is traversed, the loop keeps the sequence(s) with the minimum partial cost, which effectively results in the optimal cluster configuration sequence for the time-window.

Algorithm 2 realizes the operation of PARTIAL_COST() and computes the entire cost including transitioning, violation of SLA, pay-backs from public CSPs, and operational overheads, for a suggested new configuration. PARTIAL_COST() takes as input the current configuration (old_cl_config), the proposed new configuration (cl_config), the estimated transition duration (tr_duration), delay (tr_delay) and overhead (tr_overhead) as well as the start time (start_time) and returns the total cost of the transition period. The additional work that a private CSP has to undertake to bring the cluster to its new suggested state is designated by the tr_delay interval. The latter corresponds to the latency of the transition and through this period, the cluster appears as operating its prior configuration (old_cl_config). When VMs are moved in and out of a configuration, they remain idle during this process –no service is provided– and tr_delay accounts for the effort required to accomplish this re-alignment of resources. As soon as tr_delay is accounted for, the transition is in progress. In this transition phase, the operating VMs of the cluster involve elements from both old and new configurations as: 1) newly introduced VMs become fully functional after the completion of the transition, and 2) VMs to be removed are released immediately after tr_delay.

**Algorithm 2 Partial Cost**

```plaintext
procedure PARTIAL_COST(old_cl_config, cl_config, tr_duration, tr_delay, tr_overhead, start_time)
  // the nodes allocated during tr_delay
  tr_cl_config = INTERSECT(cl_config, old_cl_config)
  // the fully functional cluster during the transition
  cost = 0
  time = start_time
  tr_end_time = start_time + tr_delay + tr_duration
  while time < tr_end_time do
    wl = workload[time]
    // workload is the array of predicted future workload
    // (operations per time unit)
    if time - start_time < transition_delay then
      p_cost = CLUSTER_CONFIG_COST(op_cl_config, cl_config, old_cl_config, wl, 0)
    else
      p_cost = CLUSTER_CONFIG_COST(cl_config, tr_cl_config, wl, tr_overhead)
    end if
    cost += p_cost
  end while
  return cost
end procedure
```

For a specific point in time, CLUSTER_CONFIG_COST() computes the operational and penalty costs incurred by possible SLA violations. Algorithm 3 takes as input the VMs currently allotted (op_cl_config), the cluster configuration (cl_config), the expected workload at this time instance (expressed in number of operations per time unit) and the transition overhead (tr_overhead). CLUSTER_CONFIG_COST() returns the operational cost of the cluster configuration during the time unit in question. To compute potential SLA violations, we use linear and support vector regression to gauge the maximum throughput of a given cluster configuration and the DROP rate (Section III). The above is accomplished by respectively invoking PREDICT_THROUGHPUT() and PREDICT_DROP(). The fraction of SLA violations accorded to VMs coming off public CSPs yields pay-backs to the private CSP. OPER_CL_COST() determines the sum of the current rental/operational cost of each node within op_cl_config depending on whether the VMs in question belong to either a public or the private CSP.

**Algorithm 3 Cluster Configuration Cost**

```plaintext
procedure CLUSTER_CONFIG_COST(op_cl_config, cl_config, workload, tr_overhead)
  total_workload = tr_overhead + workload
  cluster_throughput = PREDICT_THROUGHPUT(cl_config)
  handled_workload = min(cluster_throughput, total_workload)
  drop = PREDICT_DROP(cluster_config)
  violations = max(total_workload - handled_workload, 0)
  // violations due to throughput
  violations += handled_workload * drop
  // violations due to DROP
  violations_per_node = violations / cl_config.nodes
  if node.belongs to public csp then:
    payback += node.sla.penalty * violations_per_node
  end if
  for node in cl_config do
    if node.nodes > 0 then
      payback += node.sla.penalty * violations_per_node
    end if
  end for
  total_penalty = app.sla.penalty * violations - payback
  if app.sla.penalty is the penalty for each SLA violation in the application
  return OPER_CL_COST(op_cl_config) + total_penalty
end procedure
```

### V. Evaluation

#### A. Our Cost Model vs. SLA-Cost Minimization

We present key evaluation results derived with a prototype that implements our suggested provisioning approach. Our system is written in Python v.2.7.5 and uses the scikit-learn library [19] for SVR computing. Table II depicts the key

<table>
<thead>
<tr>
<th>CSP</th>
<th>CPU cores</th>
<th>CPU freq (GHz)</th>
<th>RAM (GB)</th>
<th>Penalty per SLA violation (in monetary units)</th>
<th>Cost (per sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>prv</td>
<td>4</td>
<td>3.2</td>
<td>8</td>
<td>–</td>
<td>35</td>
</tr>
<tr>
<td>prv</td>
<td>2</td>
<td>2.4</td>
<td>6</td>
<td>–</td>
<td>40</td>
</tr>
<tr>
<td>pub1</td>
<td>4</td>
<td>2.4</td>
<td>3</td>
<td>0.3</td>
<td>55</td>
</tr>
<tr>
<td>pub1</td>
<td>4</td>
<td>2.4</td>
<td>4</td>
<td>0.15</td>
<td>60</td>
</tr>
<tr>
<td>pub2</td>
<td>4</td>
<td>3.2</td>
<td>8</td>
<td>0.25</td>
<td>65</td>
</tr>
<tr>
<td>pub2</td>
<td>4</td>
<td>3.4</td>
<td>8</td>
<td>0.2</td>
<td>75</td>
</tr>
</tbody>
</table>
features of VMs used in experiments along with respective costs and SLA violation penalties as advertised by public CSPs.

We set the penalty for each SLA violation of the application running to 0.3 monetary units and the time-window size to 60 secs. In the beginning of every experiment, a cluster consists of 2 VMs from the private CSP. For simplicity, we add/remove 1 VM during each transition every 60 secs. We investigate the following:

- how our cost model (A1) fares with the conventional SLA-cost minimization approach,
- the effect that the transition cost has on provisioning and
- how our approach reacts in short/long workload spikes.

The workload of the experiment was created using the YCSB and we employed epochs demonstrating periodic behavior. Every such epoch has length of 600 secs (i.e., top curves in Figs. 4, 5, and 6). Within every epoch, the workload is gradually increased and remains high for about 250 secs. Then, it is decreased abruptly until a short spike is met on the 300th sec. After the spike the workload’s trend remains unchanged until another spike is met on the 480th sec. Then workload gradually decreases until the end of the epoch. The random generators used to produce the workload follow uniform distributions. Although, we run numerous and lengthy experiments, we only report representative results from a specific range of 1,000 secs for readability purposes.

We use RHC as the main framework for provisioning and we compare our cost model with that of the widely-used SLA-cost minimization approach [3], [16]. Both techniques are deployed in a private/public CSP infrastructure and we track the number of allocated VMs over time for the execution of a synthetic workload, which consists of a diverse number of GET operations per sec. We also monitor the following accrued costs as reported by the (YCSB) client: 1) the penalty cost of the SLA violations, 2) the operational cost of the private CSP, 3) the lease cost of VMs rented from a public CSP, 4) the penalty payback, and 5) the transition cost. The SLA-cost minimization approach involves only the penalty cost due to SLA violations and it does not include cluster operational costs, pay-backs from public CSPs as well as transition costs (i.e., tr_overhead is 0).

Figs. 4 and 5 depict the number of VMs used by the SLA-cost minimization approach and our provisioning approach respectively. Fig. 4 shows that the SLA-cost minimization approach tends to allocate more VMs to handle the workload and to maximize QoS. The conventional SLA-cost minimization approach does not take into account the operational cost of the VMs in the cluster and thus, often chooses configurations with the highest performance capacity. This approach appears to “encourage” changes in cluster configurations, as there is no consideration for transitional costs.

In contrast, Fig. 5 shows that our approach requires fewer VMs for most of the time and releases them as soon as they are not needed to reduce operational cost. The transition cost makes our approach more conservative to changes. In our approach, there are only 4 encountered configuration changes compared to 13 in the SLA-cost minimization method. By considering operational/transition costs as well as pay-backs, our approach tries to balance performance capacity on the one hand, and the investment in new cluster configurations with more/fewer resources on the other. This is why our approach uses only 2 VMs to handle the workload in time interval 240−1,000 while the conventional SLA-cost minimization method exploits most of available VMs in the experiment. Note that in our approach the last 2 VMs are different than the initial. In fact, they both belong to public CSPs and the reason the algorithm does not decide to move to another cluster configuration is because the payback from these VMs balances the total cost. Also, our approach decides the first transition before the SLA-cost minimization approach. This is due to the fact that the node selected by our approach is the first that belongs to a public CSP and hence, the payback from this node results in a better solution. In the SLA-cost minimization approach this does not happen since the workload can be handled by the initial cluster configuration since it is fairly
low (less than 11,000 operations per sec).

When it comes to costs, the SLA-cost minimization approach is on average 475% more expensive than our approach just for the limited 1,000 secs of observation. Since the SLA-cost minimization approach ignores the transition cost, it falls into many situations where the cluster is overloaded, hence the current throughput drops dramatically. The reduction of the throughput in the SLA-cost minimization approach results in 41% fewer operations completed in the reported period.

The above results clearly show that the penalty minimization of the SLA violations does not lead to the minimization of the total cost. As our approach can better capture the actual costs involved in the execution of workloads, it is of substantial value to NoSQL-database owners.

B. The Effect of Transition Cost

We next evaluate how transition cost affects the cluster configuration changes. When a cluster that runs a NoSQL–database allocates or deallocates VMs, shards need to be transported. The overhead of this process is the transition cost, which is a key factor for provisioning since it indirectly affects the total cost of the private CSP. Fig. 6 depicts the results of the same experiment as that of Fig. 5 but with lower transition cost. Here, we set the $tr_{	ext{overhead}}$ of Table 1 at 90% less than the corresponding values of the first experiment. We find that the number of transitions increases from 4 (in Fig. 5) to 16 as transition costs decrease. In the original experiment, high-transition configuration changes are not encouraged by our approach, as the potential benefits are less than the incurred cost. More specifically, the provisioner decides to release and acquire a VM continuously after 240 secs since the transition cost has been reduced and the cluster can handle the workload with only 1 VM.

With lower transition costs, our RHC–based provisioning becomes more aggressive in tracking even rapidly changing workload trends. For an effective transition to occur, the length of the time-window used must be longer than the respective transition cost. NoSQL-databases occasionally present varying transition costs. When these transition costs are also high, these systems are less less effective at handling rapidly changing workloads. Hence, the transition cost is a key factor when using a NoSQL-database as it is equally critical to throughput and DROP attained when the workload displays abrupt variations.

C. Long vs. Short Workload Spikes

In this experiment, we investigate how our RHC-based approach compares with SLA-cost minimization when there are spikes in the workload. Spikes are often met whenever the user requests are increased for some reason (i.e., flash crowds).

Fig. 7 shows a workload featuring a long and a short spike at time intervals 120–420 and 720–735, respectively. The figure also depicts how our RHC-based provisioning and the SLA-cost minimization approaches behave in both instances. Our approach handles the long spike by adding VMs while it essentially ignores the short spike. During the short spike, the cluster does not move to another configuration to handle this short-lived demand as the total cost for a possible transition is greater than the cost of inaction. At the end of the long spike, the SLA-cost minimization method ends up having 5 out of 6 of the available VMs, which leads to a long deallocation period as deallocations are not instant. Hence, unnecessary VMs continue to be allocated for some time after the spike ceases which results in higher operational cost. During the short spike, the conventional SLA-cost minimization method attempts to allocate additional resources being oblivious of what lies ahead. In this specific instance (i.e., time interval 720–735), the transition costs involved are of similar length to the spike in question. Thus, additional VMs become functional beyond the time at which the spike ends yielding a resource thrashing situation. Our RHC-based provisioning avoids such thrashing because it continually evaluates the total cost of every feasible sequence of cluster configurations in a time-window and picks the best. As for the cost, the SLA-cost
minimization approach costs 92% more than our RHC-based provisioning approach for the observed period.

VI. RELATED WORK

Resource provisioning for cloud-based systems has recently attracted considerable attention. Efforts in [12], [13] attempt to address the problem through the use of queueing theory and respective model building. As cloud systems are inherently complex, involve parallel and concurrent aspects and are built on heterogeneous environments, such queueing theory models are difficult to extend and quickly become intractable. The extensive use of caching and locking policies further exacerbates matters [20]. Sharma et al. [5] present a system that statically searches for best allocation scenarios and then picks one that minimizes migration costs. The work also advocates for the adoption of performance model building through profiling. Roy et al. [4] presented an RHC-based approach that minimizes the operational cost of a host–cloud while satisfying all SLAs. However, price variation for resources as well as penalty pay-backs from public CSPs are not taken into account in the used price model. Goudarzi et al. [12] outline a heuristic approach to minimize the total energy cost of a cloud computing system while keeping the SLA-incurred costs low. Although this work appears to incorporate multiple costs into the minimization problem, the use of queueing theory models entails issues similar to those mentioned above. Barker et al. [3] presented a migration approach for multi-tenant databases that utilize a throttling controller; the latter aims to dynamically vary the migration speed to avoid SLA violations due to the transition cost imposed by a migration. To the best of our knowledge there is no work that combines penalty pay-back from public CSPs, a critical aspect from the private CSP’s point of view, and only a few [3]–[5] take into account the transition cost. Our work also takes a holistic approach and addresses resource provisioning through the occasional leasing of public resources in a way that minimizes total cost for the private CSP.

VII. CONCLUSIONS & FUTURE WORK

In this paper, we investigate how NoSQL-databases running on private Cloud-Service Providers (CSPs) could be partially “tossed out” to opportunistically exploit resources available from public CSP counterparts. Such collaborative auto-scaling helps both minimize total cost for the private CSP–hosted application and more flexibly address QoS-requirements. We presented a resource provisioning approach based on look-ahead optimization that leads to lower CSP costs for a limited time-window while considering how to best transform the utilized virtual infrastructure over time. We identify key factors that contribute to the CSP aggregate cost and propose a cost model that accounts for both direct and indirect penalties to avoid SLA violations for the hosted-application(s). Our evaluation demonstrates the benefits of our cost model over the conventional approach of simply minimizing SLA cost with reported gains of up to 47.5% for the conducted experiments. Moreover, we show that the use of a look-ahead optimization technique helps avoid resource allocation thrashing when the workload changes rapidly. We plan to investigate the relaxation of the accuracy of the used predictor, examine the respective ramifications and ascertain the role introduced errors may have in workload estimation.

Acknowledgment: This work has been partially supported by iMarine and Sucre EU-FP7 projects, ERC Starting Grant #279237 and a THALES grant co-financed by EU-ESF and the Greek NSFR “Education and Lifelong Learning” Program.

REFERENCES