A Data Quality-aware Cloud Service based on Metaheuristic and Machine Learning Provisioning Algorithms

CONFERENCE PAPER · JANUARY 2015

DOI: 10.1145/2695664.2695753

3 AUTHORS:

Dimas Nascimento
Universidade Federal de Campina Grande (U…
4 PUBLICATIONS 0 CITATIONS
SEE PROFILE

Carlos Eduardo Pires
Universidade Federal de Campina Grande (U…
34 PUBLICATIONS 34 CITATIONS
SEE PROFILE

Demetrio Gomes Mestre
Universidade Federal de Campina Grande (U…
4 PUBLICATIONS 1 CITATION
SEE PROFILE

A Data Quality-aware Cloud Service based on Metaheuristic and Machine Learning Provisioning Algorithms

Dimas C. Nascimento
Federal University of Campina Grande, Paraíba, Brazil
Federal Rural University of Pernambuco, Brazil
dimascnf@copin.ufcg.edu.br

Carlos Eduardo Pires
Federal University of Campina Grande, Paraíba, Brazil
cesp@dsc.ufcg.edu.br

Demetrio Gomes Mestre
Federal University of Campina Grande, Paraíba, Brazil
demetrio@copin.ufcg.edu.br

ABSTRACT

Cloud Computing as a service has become a topic of increasing interest. The outsourcing of duties and infrastructure to external parties became a crucial concept for many business models. In this paper we discuss the design and experimental evaluation of provisioning algorithms, in a Data Quality-aware Service (DQaS) context, that enables dynamic Data Quality Service Level Agreements (DQSLA) management and optimization of cloud resources. The DQaS has been designed to respond effectively to the DQSLA requirements of the service customers, by minimizing SLA penalties and provisioning the cloud infrastructure for the execution of data quality algorithms. An experimental evaluation of the proposed provisioning algorithms, carried out through simulation, has provided very encouraging results that confirm the adequacy of these algorithms in the DQaS context.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Data Quality, Cloud Computing

Keywords

Data Quality, Cloud Computing, Provisioning, Machine Learning, Metaheuristic

1. INTRODUCTION

Data Quality Monitoring (DQM) is the continuous process that evaluates a set of data to determine if they meet the planning objectives of a project. Any DQM life cycle will include an evaluation (assessment) phase, which intends to access and process the data according to data quality objectives. In this step, each data quality objective may generate many data quality rules which are mapped to data quality algorithms. Depending on the complexity of these algorithms and the amount of data that need to be processed, the process of DQM may require a large amount of computational resources.

In practice, business managers may prefer to outsource the overall process of data storage and continuous monitoring of data quality, due to either operational or financial reasons. Nowadays, hardware- and service-level outsourcing is usually done by using cloud Computing technologies. Cloud Computing has recently emerged as a computing platform with reliability, ubiquity availability in focus [1] mainly by utilizing computing as an on demand service [10] and simplifying the time-consuming processes of hardware provisioning, hardware purchasing and software deployment. Cloud services are applications or services offered by means of cloud Computing [1]. Therefore, by adopting cloud services, business managers are considering to take advantage of the economic benefits offered by maintaining parts of its IT resources and tasks by a cloud service provider [10].

Cloud Computing provides strong storage, computation and distributed capability in support of Big Data processing. In this scenario, we propose a **Data Quality-aware Service** (DQaS) architecture in order to provide continuous monitoring of cloud databases, which relies on Data Quality Service Level Agreements (DQSLAs) established between service consumers and a DQaS provider. In order to tackle strict DQSLA requirements and/or big data monitoring, we intend to propose and evaluate different provisioning algorithms, which uses the advantage of a cloud infrastructure to perform dynamic allocation of computational resources and the execution of data quality algorithms in a distributed manner.

The main contributions of this paper are: i) the background motivation and formalization of a Data Quality SLA (Section 2); ii) the proposition of a Data Quality-aware Service architecture (Section 3); iii) the application of machine learning and heuristic algorithms in the context of the DQaS infrastructure provisioning (Section 4); iv) the DQaS cost model (Section 5) and the evaluation of the proposed provisioning algorithms according to the adopted cost model (Section 6).

2. DATA QUALITY SLA

Quality-of-Service (QoS) refers to a set of qualities or characteristics of a service, such as availability, security, response-time, throughput, latency, reliability, and reputation. Such qualities are of interest for service providers and service consumers alike [1]. The agreement between the customer and the service provider, known as the Service Level Agreement (SLA), describes agreed service functionality, cost, and qualities [4].

A SLA is an agreement regarding the guarantees of a service. It defines mutual understandings and expectations of a service between the service provider and service consumers. It consists of sections describing the commitments to service quality and service levels that the service provider must guarantee [1].

The difference between hardware service level management and Data Quality service level management is in the perceived variability in specifying what is meant by *acceptable levels of service* [8] and how to represent these *levels of data quality acceptability*. In other words, in practice it is a challenge to express a Data Quality Service Level Agreement (DQSLA). Therefore, we state the necessity of specifying a formal structure that is dynamic and flexible enough to express service customer's data quality requirements as well as service providers obligations. To this end, we formalize the notion of a DQSLA, by enabling the involved parties to express their expectations regarding informations such as data quality dimensions, parameters of the data quality algorithms, datasets to be analyzed, overall efficiency of the process, and so on.

We define a Data Quality SLA as a 9-tuple:

DQSLA = $\langle R_{id}, DQ_{dim}, M_{rules}, \Delta ts_{valid}, |\Delta D|_{thr}, T_{res}, R_{method}, SLA_{penalties}, Init_{exec} \rangle$, where:

R_{id} is a unique resource identifier of the dataset (or datasets involved) to be monitored;

DQ_{dim} is the data quality dimension [11];

M_{rules} represents the details (parameters of the data algorithms) used to evaluate R_{id};

Δts_{valid} is the timestamp interval in which the DQSLA is valid;

$|\Delta D|_{thr}$ is the amount of changes in the dataset (R_{id}) that will trigger a new execution of a data quality algorithm by the service to evaluate the quality of R_{id} as specified by the M_{rules} parameter;

T_{res} is the expected efficiency of a single evaluation (execution of a data quality algorithm) of the dataset R_{id};

R_{method} is the method (Online or Historical) of reporting the subsequent results of the evaluations of the dataset R_{id};

$SLA_{penalties}$ defines the penalties for the cases that the defined restriction time (T_{res}) is not met by the service;

$Init_{exec}$ is a boolean flag that indicates if the dataset (R_{id}) must be initially evaluated.

3. PROPOSED DQAS ARCHITECTURE

As mentioned earlier, the adoption of cloud Computing for databases and data services introduces a variety of challenges, such as strict SLA requirements and Big Data processing. To leverage elastic cloud resources, scalability is a fundamental architectural design trait for cloud databases. In the DQaS context, the service should have the means to process the DQSLA's inputs and estimate the right amount of resources that should be allocated to honor the service

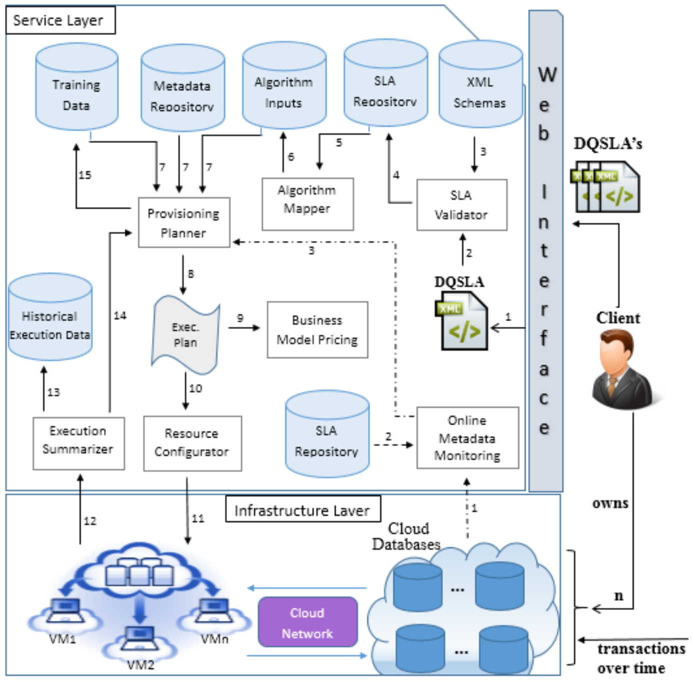

Figure 1: Data Quality-aware Service Architecture.

commitments specified at the DQSLA. In practice, the size of the datasets that need to be monitored and the complexity of the data quality algorithms that need to be executed by the DQaS may vary over time. Hence, the amount of resources needed to honor the DQSLAs may also vary notably over time. In order to tackle the infrastructure provisioning problem, we propose provisioning algorithms that are further discussed in Section 4.

Another crucial aspect of the DQaS architecture is to maximize the efficacy on the usage of the available resources while minimizing the provider bill. In other words, the following requirements have to be met: i) guarantee that the customers' DQSLAs requirements are met; and ii) optimize the resource utilization in meeting the above requirements.

The proposed DQaS architecture is shown in Figure 1. In the step 1, a customer, that initially owns one or more cloud databases, submits the input parameters of a DQSLA via a web form or a XML document to the DQaS web interface. In step 2, the received DQSLA is then validated by the *SLA Validator* module using a XML Schema selected (step 3) according to the value of the parameters DQ_{dim} and M_{rules} of the DQSLA. In step 4, if the DQSLA structure and content are properly validated by the selected XML Schema, the DQSLA is stored as a set of inputs for a data quality algorithm (step 6).

Then, if the DQSLA parameter $Init_{exec}$ is set to true, in the step 7 the *Provisioning Planner* module gathers data from the data quality algorithm (derived from the DQSLA) inputs, the cloud databases metadata and (optionally) from a training data repository in order to estimate the ideal infrastructure to execute a data quality algorithm and meet the DQSLA restriction time, i.e., by providing an execution time equal or lower than the T_{res} value of the recieved DQSLA. As an output of the step 7, an execution plan (which consists of a grid configuration) is created. The execution plan specification is used by both the *Business Model* module (in order to further aggregate customers bills) and

Table 1: The infrastructure provisioning estimation problem.

Find	Cl_i
Over	$(N \times \gamma) = \{Cl_1, Cl_2, \ldots, Cl_{n\times k}\}$
for Minimizing	$(T^e_{res} - ExecTime(e))$
Subject to	$(T^e_{res} - ExecTime(e) \geq 0)$

the *Resource Configurator* module, in order to allocate a grid of virtual machines to execute the data quality algorithm in a distributed manner.

In step 13, after the execution of the data quality algorithm is completed, the *Execution Summarizer* module aggregates and summarizes the execution data in order to populate an historical execution data repository and forward the summarized execution data to the *Provisioning Planner* module in order to: i) update the training data repository and ii) (optionally) provide an online update of a report that is shown by the web interface of the DQaS.

Another possible flow of tasks (displayed by dashed lines in the architecture) performed by the DQaS is triggered when the amount of changes (inserts, updates or deletes) in a dataset being currently monitored exceeds the $|\Delta D|_{thr}$ parameter value specified by the client in the DQSLA. When this scenario is detected by the *Online Metadata Monitoring* module, the *Provisioning Planner* is notified and uses a set of input data to create an execution plan and evaluate the changed dataset, as explained earlier in this section.

3.1 Configuration Class

The service's resource requirements are fulfilled by the physical or virtual computational nodes upon which the data quality algorithms are executed.

In practice, the configuration class estimated by the *Provisioning Planner* module is a virtual machine cluster composed by a pair ⟨*Number of VM's* , *VM_Configuration*⟩. Let $N = \{N_1, N_2, \ldots, N_n\}$ be the amount of VM's that can be allocated for an execution, $\gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_k\}$ the available VM configurations and a Configuration Class (Cl_l) be a pair $<N_i, \gamma_j>$. Then, $Cl = (N\times\gamma) = \{Cl_1, Cl_2, \ldots, Cl_{n\times k}\}$ represents the set of all possible configuration classes that can be chosen by the *Provisioning Planner* module. In order to facilitate the proposition of the provision algorithms, we use Cl as a partially ordered set that is paired by a relation defined as follows:
I.$\forall Cl_i, Cl_j \in Cl : (i \geq j) \Rightarrow Pr(Cl_i) \geq Pr(Cl_j)$
II. $\forall Cl_i, Cl_j \in Cl :$
$((i < j) \wedge (Pr(Cl_i) = Pr(Cl_j))) \Rightarrow Cl_i.N < Cl_j.N$
, where Pr(Cl_l) denotes that the processing capacity of Cl_l and $Cl_l.N$ denotes the number of nodes (VM's) of Cl_l.

4. PROVISIONING ALGORITHMS

We are now able to formalize the problem that is tackled by the *Provisioning Module* in the proposed DQaS architecture. For a single execution e of a data quality algorithm, we want to estimate a proper class Cl_i that is able to minimize the difference between the restriction time defined in the DQSLA that triggered the execution e (T^e_{res}) and the execution time of e ($ExecTime(e)$), as stated in Table 1.

4.1 Data Quality Scope

Although the core ideas of the provisioning algorithms proposed in this paper may also be used for other data quality tasks, we limit the scope of this paper to propose provisioning algorithms that are adjusted according to the execution results of data quality algorithms for deduplication tasks. The process of deduplication consists in identifying duplicated entities among a single dataset [3]. The inputs of a deduplication algorithm triggered by a DQSLA S_{Id} may be summarized using the following vector notation:
$\hat{s_{Id}} = \langle s_{Id1}, s_{Id2}, s_{Id3}, s_{Id4}\rangle = \langle Comparisons(s_{Id}),$
$ComparisonCost(s_{Id}), T^{s_{Id}}_{res}, CurrExecClass(s_{Id})\rangle$
, where: $Comparisons(s_{Id})$ is the amount of comparisons that will be executed among the entities, which depends on the size of the dataset and on the deduplication algorithm that is being executed (note that these informations can be both derived from the DQSLA parameters); in practice, the number of comparisons may be theoretically [3] estimated based on the dataset size and on the distribution of its data;
$ComparisonCost(s_{Id})$ is the cost of a single comparison between two entities, which depends on the complexity of the similarity function(s) used by the deduplication algorithm and on the amount and the size of entity attributes that need to be compared;
$T^{s_{Id}}_{res}$ is the input parameter of the S_{Id} DQSLA; and
$CurrExecClass(s_{Id})$ is the configuration class used for the most recent execution triggered by S_{Id} (or Nil if S_{Id} has not triggered an execution yet).

Regarding the training data, its content is composed by a set of 4-tuples using the following vector notation:
$\hat{b_j} = \langle b_{j1}, b_{j2}, b_{j3}, b_{j4}\rangle = \langle Comparisons(b_j),$
$ComparisonCost(b_j), ExecTime(b_j), Class(b_j)\rangle$
, where ExecTime(b_j) is the execution time of the deduplication algorithm (that generates Comparisons(b_j) and Comparison Cost(b_i)) using Class(b_j) as an infrastructure class.

4.2 Hill Climbing Estimation

The *Provisioning Planner* module uses a metaheuristic algorithm (*Hill Climbing*) to adjust the configuration class (Cl_i) used to execute the data quality algorithms over subsequent executions triggered by a DQSLA (s_{Id}). We used a Hill Climbing approach, instead of A* or Best-first Search, because the *Provisioning Module* can not evaluate all available configuration classes, due to the high computational costs, to decide which option is able to minimize the objective function ($T^e_{res} - ExecTime(e)$).

The pseudo code of the Hill Climbing is represented by the Algorithm 1. Initially (lines 2 and 3), the algorithm checks if the DQaS has not performed an execution triggered by s_{Id} yet. If so (lines 4 to 6), the algorithm creates an initial configuration class (Cl_I), schedules an execution plan based on Cl_I and updates the training data (TB) using the results of the execution plan. Otherwise (lines 8 to 18), the algorithm checks if the previous execution class used for the s_{Id} DQSLA is not considered a good solution (line 8). If so, the algorithm performs a tweak on the previous configuration class used for the s_{Id} DQSLA (line 10), updates the training base using the results of the execution using Cl_T and checks (lines 13, 14 and 17) if the adjusted configuration class (Cl_T) should replace (line 16 or 18) the previous configuration class (Cl_P). Finally, if both cases (line 3 or line 8) are not evaluated as true, i.e., the previous configuration class (Cl_P) used for s_{id} is considered a good solution, then the algorithm creates an execution plan using Cl_P (line 20).

Note that there are two specific lines (5 and 10) in which the Hill Climbing algorithm executes an interchangeable call.

In practice, it means that each of these lines may be replaced by different algorithm calls (for the generation of an initial configuration class or the adjustment of a new configuration class for a DQSLA) in order to generate different hill climbing algorithms.

```
Algorithm 1: HillClimbingEstimation
input : $s_{Id}$: a DQSLA id
        $thr$: a (positive) minimum threshold for tweak the current solution
        $L_{index}$: the index of the last available configuration class
        $TB$: a training base
1  begin
2      $Cl_P \longleftarrow CurrExecClass(s_{Id})$
3      if ($Cl_P = Nil$) then
4          // interchangeable call
5          $Cl_I \longleftarrow InitialSolutionClass(S_{Id}, L_{index})$
           $CurrExecTime(s_{Id}) \longleftarrow ExecPlan(s_{Id}, Cl_I)$
           $CurrExecClass(s_{Id}) \longleftarrow Cl_I$
6          $TB \longleftarrow TB \cup$
           $\langle Comparisons(s_{Id}), ComparisonCost(s_{Id}),$
7          $CurrExecTime(s_{Id}), Cl_I \rangle$
8      else if (($T^{s_{Id}}_{res} - PrevExecTime(s_{Id})) > thr$ or
       $PrevExecTime(s_{Id}) > T^{s_{Id}}_{res}$) then
9          // interchangeable call
10         $Cl_T \longleftarrow TweakSolution(Cl_P, s_{Id}, L_{index})$
           $CurrExecTime(s_{Id}) \longleftarrow ExecPlan(s_{Id}, Cl_T)$
11         $TB \longleftarrow TB \cup$
           $\langle Comparisons(s_{Id}), ComparisonCost(s_{Id}),$
12         $CurrExecTime(s_{Id}), Cl_T \rangle$
13         if ($|CurrExecTime(s_{Id}) - T^{s_{Id}}_{res}| <$
           $|PrevExecTime(s_{Id}) - T^{s_{Id}}_{res}|$) then
14             if (($PrevExecTime(s_{Id}) < T^{s_{Id}}_{res}$) and
15             ($CurrExecTime(s_{Id}) < T^{s_{Id}}_{res}$)) then
16                 $CurrExecClass(s_{Id}) \longleftarrow Cl_T$
17             else if ($PrevExecTime(s_{Id}) > T^{s_{Id}}_{res}$) then
18                 $CurrExecClass(s_{Id}) \longleftarrow Cl_T$
19     else
20         $ExecPlan(s_{Id}, Cl_P)$
```

4.3 Metaheuristic and Heuristic Algorithms

In this section, we present 3 heuristic algorithms that are used to adjust (tweak) a current solution (a configuration class) in the Hill Climbing algorithm (line 10). The SlicedTweakSolution algorithm (Algorithm 2) performs a random adjust (scaling up or scaling down) on the input configuration class (Cl_i), depending on the difference between CurrExecTime(s_{Id}) and $T^{s_{Id}}_{res}$. Moreover, the Sliced TweakSolution uses two persistent variables (InferiorIndexLimit(s_{Id}) and SuperiorIndexLimit(s_{Id})) to avoid performing adjustments on the input class (Cl_i) beyond necessity.

The BinaryTweakSolution (Algorithm 3) algorithm performs a binary search-based adjustment (scaling up or scaling down) on the input configuration class (Cl_i), depending on the difference between $CurrExecTime(s_{Id})$ and $T^{s_{Id}}_{res}$.

Lastly, the TunableTweakSolution (Algorithm 4) algorithm adjusts (scaling up or scaling down) the input configuration class according to a tunable input parameter (*ScaleFactor*). The main purpose of this algorithm is to be used in conjunction with machine learning-based initial solutions (Section

```
Algorithm 2: SlicedTweakSolution
input : $Cl_i$: an input configuration class
        $s_{Id}$: a DQSLA id
        $LastClassIndex$: the index of the last available configuration class
output: $Cl_o$: an output configuration class
1  persistent InferiorIndexLimit($s_{Id}$)
2  persistent SuperiorIndexLimit($s_{Id}$)
3  begin
4      if ($InferiorIndexLimit(s_{Id}) = Nil$) then
5          $InferiorIndexLimit(s_{Id}) \longleftarrow 1$
6      if ($SuperiorIndexLimit(s_{Id}) = Nil$) then
7          $SuperiorIndexLimit(s_{Id}) \longleftarrow LastClassIndex$
8      if ($CurrExecTime(s_{Id}) > T^{s_{Id}}_{res}$) then
9          // scaling up
10         $Cl_R \longleftarrow randomClass(i + 1,$
11         $SuperiorIndexLimit(s_{Id}))$
12         $InferiorIndexLimit(s_{Id}) \longleftarrow i + 1$
13         return $Cl_R$
14     else if ($CurrExecTime(s_{Id}) < T^{s_{Id}}_{res}$) then
15         // scaling down
16         $Cl_R \longleftarrow$
           $randomClass(InferiorIndexLimit(s_{Id}),$
17         $i - 1)$
18         $SuperiorIndexLimit(s_{Id}) \longleftarrow i - 1$
19         return $Cl_R$
20     return $Cl_i$
```

4.4), since it is expected that the machine learning algorithms are able to provide ideal (i.e., that approximates $T^e_{res} \simeq$ ExecTime(e)) or near ideal initial configuration classes, upon which will be only necessary to perform small adjustments over time.

4.4 Machine Learning Algorithms

In this section, we present two Machine Learning (ML) algorithms that can be used to estimate the initial solution (configuration class) of the Hill Climbing algorithm (line 5). The ML algorithms perform an initial estimation of the configuration classes based on a training base that is structured as defined in Section 4.1. The $kNearestNeighbors$ algorithm (Algorithm 5) behaves in the following way: given an input DQSLA vector $\hat{S_{Id}}$ (Section 4.1), the algorithm selects from the training base the top K nearest vectors and their respective configuration classes, according to a similarity measure (using Eq. (2.a) or Eq. (3.a), depending on the value of the *SimFunction* parameter), and selects the most frequent configuration class amongst them.

In turn, the *SelectPrototype* algorithm (Algorithm 6) aims to generate a representative class index amongst the configuration classes of the top K nearest vectors in the training base (using Eq. (2.a) or Eq. (3.a), depending on the value of the *SimFunction* parameter) with respect to an input DQSLA vector $\hat{S_{Id}}$. In this paper, we use the following aggregation modes: median and weighted geometric mean (Eq. 1.a). The aggregations are used as follows: i) **Median**: the median of the K nearest neighbors class indexes; and ii) **wGeomMean**: the weighted geometric mean of the K nearest neighbors class indexes. We used a fixed value for the weighting parameter (w_i) that is defined in Eq. (1.b). Thus, from Eq. (1.a) and Eq. (1.b), we can derive Eq. (1.c).

Algorithm 3: BinaryTweakSolution

```
   input  : $Cl_i$: an input configuration class
            $s_{Id}$: a DQSLA id
            $LastClassIndex$: the index of the last
            available configuration class
   output: $Cl_o$: an output configuration class
 1 begin
 2     if ($CurrExecTime(s_{Id}) > T_{res}^{s_{Id}}$) then
 3         // scaling up
 4         $o \longleftarrow \left\lceil \frac{i + LastClassIndex}{2} \right\rceil$
 5         return $Cl_o$
 6     else if ($CurrExecTime(s_{Id}) < T_{res}^{s_{Id}}$) then
 7         // scaling down
 8         $o \longleftarrow \left\lfloor \frac{1+i}{2} \right\rfloor$
 9         return $Cl_o$
10     return $Cl_i$
```

Algorithm 4: TunableTweakSolution

```
   input  : $Cl_i$: an input configuration class
            $s_{Id}$: a DQSLA id
            $LastClassIndex$: the index of the last
            available configuration class
            $ScaleFactor$: a factor used to adjust the
            input configuration class
   output: $Cl_o$: an output configuration class
 1 begin
 2     if ($CurrExecTime(s_{Id}) > T_{res}^{s_{Id}}$) then
 3         // scaling up
 4         $o \longleftarrow i + \left\lceil \frac{LastClassIndex}{ScaleFactor} \right\rceil$
 5         return $Cl_M$
 6     else if ($CurrExecTime(s_{Id}) < T_{res}^{s_{Id}}$) then
 7         // scaling down
 8         $o \longleftarrow i - \left\lfloor \frac{LastClassIndex}{ScaleFactor} \right\rfloor$
 9         return $Cl_o$
10     return $Cl_i$
```

$$wGeomMean(\hat{\mathrm{v}}) = \left(\prod_{i=1}^{n} \mathrm{v}_i^{w_i}\right)^{\frac{1}{\sum_{i=i}^{n} w_i}} =$$

$$= \exp\left(\frac{\sum_{i=1}^{n} w_i \ln \mathrm{v}_i}{\sum_{i=1}^{n} w_i}\right) \tag{1a}$$

$$w_i = i^2 \tag{1b}$$

$$wGeomMean(\hat{\mathrm{v}}) = \exp\left(\frac{\sum_{i=1}^{n} i^2 \ln \mathrm{v}_i}{\sum_{i=1}^{n} i^2}\right) \tag{1c}$$

Algorithm 5: kNearestNeighbors

```
  input  : $s_{Id}$: a DQSLA id
           $TB$: a training base
           $k$: quantity of Neighbors
           $SimFunction$: a similarity function
           {EuclidianDistance, CosineSimilarity}
  output: $Cl_o$: an output configuration class
1 begin
2     $kNearestNeighbors[] \longleftarrow$
      $kNearestNeighborsClasses(TB, \hat{s_{Id}}, k, SimFunction)$
3     return $MostFrequentClass(kNearestNeighbors)$
```

Algorithm 6: SelectPrototype

```
   input  : $s_{Id}$: a DQSLA id
            $TB$: a training base
            $k$: quantity of Neighbors
            $AggregationMode$: the chosen aggregation
            mode {Median, wGeomMean}
            $SimFunction$: a similarity function
            {EuclidianDistance, CosineSimilarity}
   output: $Cl_o$: an output configuration class
 1 begin
 2     $kNearestNeighbors[] \longleftarrow$
       $kNearestNeighborsClasses(TB, \hat{s_{Id}}, k, SimFunction)$
 3     if ($AggregationMode = Median$) then
 4         // ascendant sorting based on the indexes
              of the classes
 5         $Sort(kNearestNeighbors)$
 6         $MedianIndex \longleftarrow \left\lfloor \frac{k}{2} \right\rfloor$
 7         return $kNearestNeighbors[MedianIndex]$
 8     else if ($AggregationMode = wGeomMean$) then
 9         // ascendant sorting based on the
              proximity of the neighbors to s_Id
10         $Sort(kNearestNeighbors)$
11         $NeighborsIndexes \longleftarrow$
           $ExtractIndexes(kNearestNeighbors)$
12         // using Eq. 1c
13         $o \longleftarrow wGeomMean(NeighborsIndexes)$
14         return $Cl_o$
```

$$kNearestNeighborClasses(TB, \hat{s_i}, k, Euclidian) =$$

$$= Top\ k\left(Class\left(\arg\min_{\hat{b_j} \in TB} z(||\hat{s_i} - \hat{b_j}||)\right)\right) \tag{2a}$$

$$\arg\min_{\hat{b_j} \in TB} z(||\hat{s_i} - \hat{b_j}||) =$$

$$= \arg\min_{\hat{b_j} \in TB} \mathrm{sqrt}\left(\left(||z(\hat{s_i})||\right)^2 + \left(||z(\hat{b_j})||\right)^2 - 2 \times (z(\hat{s_i}) \cdot z(\hat{b_j}))\right) \tag{2b}$$

$$\hat{s_i} \cdot \hat{b_j} = \sum_{k=1}^{3} s_{ik} \times b_{jk} \tag{2c}$$

$$||\hat{v_i}|| = \sqrt{\hat{v_i} \cdot \hat{v_i}} = \sqrt{\sum_{k=1}^{3} (v_{ik})^2} \tag{2d}$$

$$z(\hat{v_k}) = \left\langle \frac{v_{k1}}{\arg\max_{\hat{b_j} \in TB} b_{j1}}, \frac{v_{k2}}{\arg\max_{\hat{b_j} \in TB} b_{j2}}, \frac{v_{k3}}{\arg\max_{\hat{b_j} \in TB} b_{j3}}, v_{k4} \right\rangle \tag{2e}$$

$$NearestNeighborClass(TB, \hat{s_i}, k, Cosine) =$$

$$= Top\ k\Bigg(Class\Bigg(\underset{\hat{b_j} \in TB}{\arg\min}\, z\Bigg(\frac{\hat{s_i} \cdot \hat{b_j}}{||\hat{s_i}|| \times ||\hat{b_j}||}\Bigg)\Bigg)\Bigg) \tag{3a}$$

$$\underset{\hat{b_j} \in TB}{\arg\min}\, z\Bigg(\frac{\hat{s_i} \cdot \hat{b_j}}{||\hat{s_i}|| \times ||\hat{b_j}||}\Bigg) =$$

$$\underset{\hat{b_j} \in TB}{\arg\min}\Bigg(\frac{z(\hat{s_i}) \cdot z(\hat{b_j})}{||z(\hat{s_i})||) \times ||z(\hat{b_j})||)}\Bigg) \tag{3b}$$

5. COST MODEL

In this section, we describe the cost model of the proposed DQaS. The model is used to evaluate the effectiveness of the proposed provisioning algorithms (Section 4) according to their capacity to minimize the costs of the service provider.

Let $C\langle ts_i, ts_f\rangle$ be the set of customers of the DQaS during a timestamp interval (ts_i, ts_f). During (ts_i, ts_f), each customer c $\in C\langle ts_i, ts_f\rangle$ may create a set of DQSLAs that is represented by $S_c\langle ts_i, ts_f\rangle$. Each DQSLA s $\in S_c\langle ts_i, ts_f\rangle$ may trigger a set of executions represented by $E_s\langle ts_i, ts_f\rangle$, during the timestamp interval (ts_i, ts_f).

Each execution $e \in E_s\langle ts_i, ts_f\rangle$ has an infrastructure cost that is influenced by the execution time (*ExecTime*), initialization time (*InitTime*) and the infrastructure price (*Price*) of the allocated VM's. Let *Init*(e) represent all the initializated and used VM's by a particular execution e. Each $VM_i^e \in Init(e)$ has a unique and sequential identifier i.

Let γ indicates the set of possible configurations for a VM, then each $VM_{il}^e \in Init(e)$ presents the same configuration ${conf_i}^e = l$ and $Price(VM_{il}^e)$ price. Let the infrastructure cost of the execution e using a cluster of VM configuration l be denoted by $VMCost^e$, then we can estimate this cost according to Eq. (4).

$$\begin{aligned} VMCost^e = &\sum_{v=1}^{|Init(e)|} (ExecTime(VM^e_{vconf^e_v} + \\ &InitTime(VM^e_{vconf^e_v})) \times Price(VM^e_{vconf^e_v}) \\ = &\sum_{v=1}^{|Init(e)|} (ExecTime(VM^e_{vl}) + \\ &InitTime(VM^e_{vl})) \times Price(VM^e_{vl}) \end{aligned} \tag{4}$$

The total cost $(Cost^e)$ of a single execution e is composed by its infrastructure costs $(VMCost^e)$ plus its penalty costs, defined as follows: $Cost^e = VMCost^e + PenaltyCost^e$.

Each execution $e \in E_s\langle ts_i, ts_f\rangle$ performed by a DQaS is subjected to a time restriction T^e_{res} that is specified at the DQSLA $s \in S_c\langle ts_i, ts_f\rangle$ that triggered the execution e. In this work, we use a SLA penalty violation model that is similar to other related works [12] and is modeled as a linear function as follows: $SLAPenalty = \alpha + \beta \times DT$. In the SLA penalty model, α represents the fixed penalty, β is the penalty rate and DT is the delay time between the execution time (*InitTime(e) + ExecTime(e)*) and the restriction time (T^e_{res}) of an execution e.

In the DQ-aware service context, the fixed penalty $(\alpha(e))$ and the penalty rate $(\beta(e))$ associated to an execution e are both specified at the DQSLA $s \in S_c\langle ts_i, ts_f\rangle$ that triggered the execution e using the $SLA_{penalties}$ parameter. Thus, we can define the $PenaltyCost^e$ of an execution e as shown in Eq. (5).

$$PenaltyCost^e = (\alpha(e) + \beta(e) \times DT(e)) \tag{5}$$

5.1 Algorithm Effectiveness

Let TB_0^k be the set of the k executions required to build the initial training base used by a machine learning provisioning algorithm (*ml_est*). Then, the cost $(ServCost^{C\langle ts_i, ts_f\rangle}_{ml\_est})$ of serving the DQaS for $|C\langle ts_i, ts_f\rangle|$ clients using a ML-based algorithm, during a timestamp interval (ts_i, ts_f), is the sum of the total cost related to the execution of data quality algorithms plus the costs for fulfilling the training base, as defined in Eq. (6).

$$\begin{aligned} ServCost^{C\langle ts_i, ts_f\rangle}_{ml\_est} = &\left(\sum_{c=1}^{|C\langle ts_i, ts_f\rangle|} \sum_{s=1}^{|S_c\langle ts_i, ts_f\rangle|} \sum_{e=1}^{|E_s\langle ts_i, ts_f\rangle|} Cost^e\right) \\ &+ \sum_{b=1}^{|TB_0^k|} VMCost^b \end{aligned} \tag{6}$$

Regarding a heuristic algorithm (*heur_est*), since it does not require a training base, we can calculate $ServCost^{C\langle ts_i, ts_f\rangle}_{heur\_est}$ as shown in Eq. (7).

$$ServCost^{C\langle ts_i, ts_f\rangle}_{heur\_est} = \left(\sum_{c=1}^{|C\langle ts_i, ts_f\rangle|} \sum_{s=1}^{|S_c\langle ts_i, ts_f\rangle|} \sum_{e=1}^{|E_s\langle ts_i, ts_f\rangle|} Cost^e\right) \tag{7}$$

6. EXPERIMENTS

In this section we discuss an assessment of the proposed provisioning algorithms we have carried out through simulation. The objective of this assessment is to verify whether the algorithms are able to estimate ideal configuration classes indexes over time that meet their DQSLAs time restrictions and optimize the allocated resources in the cloud. The evaluated provisioning algorithms, which are generated by modifying the interchangeable calls of the *HillClimbingEstimation* algorithm, are shown in Table 3.

We developed an execution model to be used in the simulation environment. The model uses the notation described in Table 3 and is presented in Eq. (8). In short, the purpose of the model is to estimate the influence of the distribution environment (number of nodes N) and the virtual machines configuration (γ) over the execution time of a deduplication task, given a speed up factor (s) that simulates the delays generated from factors such as cloud network communications and the load balancing problem [3] that usually occur in a distributed execution. Moreover, the η parameter of the model is empirically estimated by performing a single serial comparison between two entities of the dataset (according to the M_{rules} DQSLA parameter) using a single vCPU virtual machine.

Table 2: Interchangeable calls of the provisioning algorithms.

alg	InitialSolutionClass	Tweak
heur#1	$\lfloor(1 + L_{index})/2\rfloor$	Binary
heur#2	randomClass(1, L_{index})	Sliced
heur#3	$\lfloor(1 + L_{index})/2\rfloor$	Sliced
heur#4	$\lfloor(1 + L_{index})/2\rfloor$	Tunable
knn#1	kNearestNeighbors(s_{Id}, TB, 1, Euc)	Tunable
knn#2	kNearestNeighbors(s_{Id}, TB, 7, Euc)	Tunable
knn#3	kNearestNeighbors(s_{Id}, TB, 1, Cos)	Tunable
knn#4	kNearestNeighbors(s_{Id}, TB, 7, Cos)	Tunable
proto#1	Prototype(s_{Id}, TB, 1, wGeoMean, Euc)	Tunable
proto#2	Prototype(s_{Id}, TB, 7, wGeoMean, Euc)	Tunable
proto#3	Prototype(s_{Id}, TB, 1, Median, Cos)	Tunable
proto#4	Prototype(s_{Id}, TB, 7, Median, Cos)	Tunable

Table 3: Execution model notation.

D	Dataset to be monitored
bk	Blocking key of the dataset D
b	Number of blocks generated by applying bk to D
$\lambda(D, b)$	Number of comparisons to be performed
η	A single comparison cost (ms) between 2 entities
γ	Number of vCPUs
N	Number of nodes (VM's).
s	Speed up factor

$$ExecTime = \begin{cases} \dfrac{\eta \times \lambda(D, b)}{N \times \gamma} & \text{if } N = 1 \\[2ex] \dfrac{\eta \times \lambda(D, b)}{s^{-1} \times N \times \gamma} & \text{if } N > 1 \end{cases} \tag{8}$$

The workload used in the experiments is described in Table 4 and the global parameters of the algorithms and equations used in the simulated environment are shown in Table 5. Lastly, we used the amazon ec2 VM pricing reference[1] in order to calculate the infrastructure costs (Eq. (4)).

Table 4: Workload Characterization.

Parameter	Value
period	1 year
#clients	100
#DQSLAS per client	random(1, 20)
|D| (number of rows)	random(100, 3×10^7)
b (number of entities per block)	100
Deduplication algorithm	Standard blocking [3]
average #executions per month	121
#executions	1452 executions

The results of the experiments are shown in Figure 2 and Figure 3. We report, respectively: the accumulated service cost ($VMCost + SLAPenalties$) over time and the accumulated number of SLA violations, generated by the algorithm of each type (*heuristic*, *knn*, and *prototype*) that provided the smallest value for the accumulated service cost.

6.1 Discussion

[1]http://aws.amazon.com/ec2

Table 5: Global parameters.

Parameter	Location	Value
thr	Algorithm 1	$0.2 \times T_{res}^{s_{Id}}$
α	Eq. (5)	\$100
β	Eq. (5)	\$0.2 per minute
$ScaleFactor$	Algorithm 4	10
γ (#vCPUs)	Section 3.1	{1, 2, 3, 4}
s ($speedupfactor$)	Eq. (8)	random(1, 3)
N	Section 3.1	{1, 2, ..., 100}
$LastClassIndex$	Algorithm{1, 2, 3, 4}	400 ($\|N\| \times \|\gamma\|$)
$\|TB_0^k\|$	Eq. (6)	252 executions

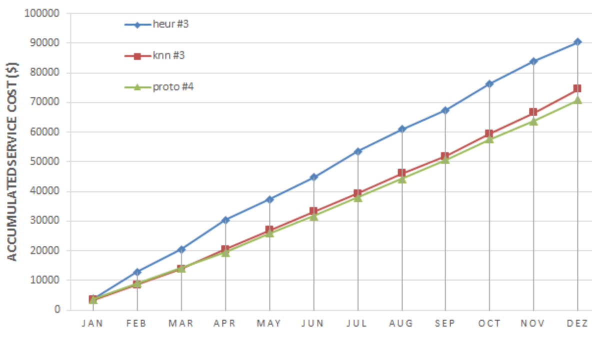

Figure 2: Accumulated service cost over time.

As expected, the ML-based provisioning algorithms generated smaller service costs over time, since they usually start the first execution of an input DQSLA using an approximately ideal configuration class. The best ML-based algorithms (*knn*#3 and *proto*#4) generated \$73,589 and \$69,958 as the accumulated service cost of the evaluated workload, respectively. Both algorithms used the cosine distance-based measure for selecting the top K most similar vectors from the training base. Surprisingly, the cosine-based 1-NN algorithm (*heur*#3) was able to provide better initial configuration estimations than its 7-NN (*heur*#4) version. Moreover, the $Median$ (used by the *proto*#4 algorithm) proved to be a good aggregation mode in order to generate prototype configuration classes.

It is important to highlight that the ML-based provisioning algorithms require an initial training base cost. The training base (254 executions) used in the experiments requires a \$926 initial cost in order to be filled. In the reported results, the training base cost was compensated due to the costs saved (compared with the costs generated by the most effective heuristic) by the usage of the most effective ML algorithms. We believe that the usage of a training base for estimating cluster configurations is a promising strategy to high workloads (a high number of estimated executions) or in cases in which the DQaS is used for a long period. Additionally, considering ML-based provisioning algorithms, the more the DQaS runs over time, the more its provisioning module becomes specialized and is able to generate lower costs for the operation of the service. This phenomenon can be noticed in Figure 3. In turn, the heuristic algorithms may also be useful in the context of light workloads (a small number of DQ algorithm executions), since they do not require initial training base costs.

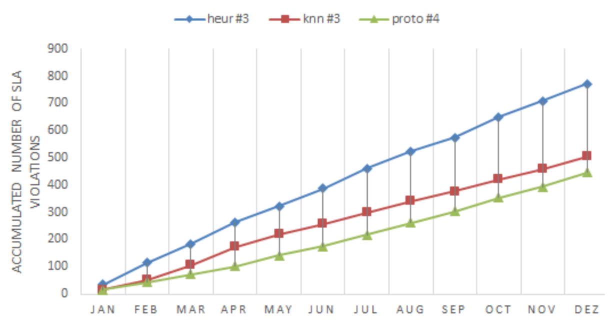

Figure 3: Accumulated number of SLA violations over time.

7. RELATED WORK

Cloud Computing has recently became a very active research area. Some works have focused on developing grid computing [9] and closed queueing network [6] models to maximize resource utilization. In [9], the authors describe node's resources and service requirements using a matrix of attributes. These works are different from our since their objective is to estimate resource allocation for static and predefined workloads, which are significantly different from the dynamic nature of a workload for which a DQaS is submitted. Machine learning has been investigated [7, 5, 2] in the context of cloud computing. Nevertheless, our work is different from [7, 5, 2] because we evaluated other machine learning algorithms and used specific workloads in the DQaS context. Lastly, provisioning algorithms for cloud Computing have also been addressed in the database context [13]. However, the main objective of the authors in [13] is to optimize resource allocation in the context of workloads represented as a set of database queries. To the best of our knowledge, we are the first to evaluate the effectiveness of provisioning algorithms in a DQaS context, i.e., by processing a workload as a set of parameters for the execution of DQ algorithms.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we formalized the notion of a data quality SLA that can be used to describe expectations and obligations of the both parts ($Clients$ and $ServiceProvider$) involved in a data quality monitoring outsourcing contract. We have also presented a DQaS architecture that we have designed to minimize SLA penalties and infrastructure costs. Moreover, we have proposed heuristic and machine learning provisioning algorithms to tackle these challenges. An initial evaluation of these algorithms, carried out through simulation, has provided very encouraging results that confirm the adequacy of our architecture. The ML-based algorithms $knn\#3$ and $prototype\#4$ generated the lowest server costs using the evaluated workload and they both use the cosine distance measure to compute vector similarities.

For future work, we are planning to carry out further experiments using a real cloud computing environment in order to test the most effective provisioning proposed algorithms. In addition, we are planning to extend the experiments in order to measure the influence of the global parameters, such as the training base size and the SLA penalties, in the results. Lastly, we plan to develop novel algorithms for the modules in the DQaS architecture that have not been addressed in this paper.

9. REFERENCES

[1] E. Badidi. A cloud service broker for sla-based saas provisioning. In *Information Society (i-Society), 2013 International Conference on*, pages 61–66. IEEE, 2013.

[2] T. Chen and R. Bahsoon. Symbiotic and sensitivity-aware architecture for globally-optimal benefit in self-adaptive cloud. In *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, Proceedings, Hyderabad, India, June 2-3, 2014*, pages 85–94, 2014.

[3] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Transactions on Knowledge and Data Engineering*, 2011.

[4] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web services on demand: Wsla-driven automated management. *IBM systems journal*, 43(1):136–158, 2004.

[5] P. Jamshidi, A. Ahmad, and C. Pahl. Autonomic resource provisioning for cloud-based software. In *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, Proceedings, Hyderabad, India, June 2-3, 2014*, pages 95–104, 2014.

[6] Y. Kouki and T. Ledoux. Scaling: Sla-driven cloud auto-scaling. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 411–414. ACM, 2013.

[7] J. Ll. Berral, R. Gavaldà, and J. Torres. Empowering automatic data-center management with machine learning. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 170–172, New York, NY, USA, 2013. ACM.

[8] D. Loshin. *The practitioner's guide to data quality improvement*. Elsevier, 2010.

[9] M. B. Reynolds, K. M. Hopkinson, M. E. Oxley, and B. E. Mullins. Provisioning norm: An asymmetric quality measure for saas resource allocation. In *Services Computing (SCC), 2011 IEEE International Conference on*, pages 112–119. IEEE, 2011.

[10] M. Schnjakin, R. Alnemr, and C. Meinel. Contract-based cloud architecture. In *Proceedings of the second international workshop on Cloud data management*, pages 33–40. ACM, 2010.

[11] F. Sidi, P. Shariat Panahy, L. S. Affendey, M. A. Jabar, H. Ibrahim, and A. Mustapha. Data quality: A survey of data quality dimensions. In *Information Retrieval & Knowledge Management (CAMP), 2012 International Conference on*, pages 300–304. IEEE, 2012.

[12] L. Wu, S. K. Garg, and R. Buyya. Sla-based resource allocation for software as a service provider (saas) in cloud computing environments. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 195–204. IEEE, 2011.

[13] P. Xiong, Y. Chi, S. Zhu, H. J. Moon, C. Pu, and H. Hacigumus. Intelligent management of virtualized resources for database systems in cloud environment. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 87–98. IEEE, 2011.