Rhone: a quality-based query rewriting algorithm for data integration.

Daniel A. S. Carvalho[1], Plácido A. Souza Neto[2], Chirine Ghedira-Guegan[3], Nadia Bennani[4], and Genoveva Vargas-Solar[5]

[1] Université Jean Moulin Lyon 3, Centre de Recherche Magellan, IAE, France
daniel.carvalho@univ-lyon3.fr
[2] Instituto Federal do Rio Grande do Norte - IFRN, Natal, Brazil
placido.neto@ifrn.edu.br
[3] Université Jean Moulin Lyon 3, LIRIS, Centre de Recherche Magellan, IAE, France
chirine.ghedira-guegan@univ-lyon3.fr
[4] LIRIS, INSA-Lyon, France
nadia.bennani@insa-lyon.fr
[5] CNRS-LIG, Grenoble, France
genoveva.vargas@imag .fr

Abstract. Data integration arises in the cloud computing as a service composition problem. Producing service compositions is computationally costly; and executing them require a considerable amount of memory, storage and computing resources. Our research focus on how enhancing the quality on data integration in a cloud context. This paper presents a rewriting algorithm named *Rhone* that addresses query for data integration. The originality of *Rhone* is the rewriting process guided by quality measures associated to data providers (services) and user preferences. The paper uses a running scenario to describe the *Rhone*'s formalization and its implementation. We also present an experimental evaluation. It shows that quality can be improved on data integration solutions. In addition, perspectives concerning our data integration approach and algorithm are presented.

Keywords: Data integration. Query rewriting. Query rewriting algorithm. Cloud computing. SLA.

1 Introduction

Current data integration implies consuming data from different data services and integrating the results according to different quality requirements related to data cost, provenance, privacy, reliability, availability, among others. Data services and data processing services can take advantage from the on-demand and *pay-as-you-go* model imposed by the cloud architecture. The quality conditions and penalties in case of its violation under which these services are delivered are defined in contracts called Service Level Agreements (SLA).

Cloud services (data services, data processing services, for instance) and the cloud provider export their SLA specifying the level of services the user can expect from them. A user willing to integrate data establishes a contract with the cloud provider guided by an economic model that defines the services she can access, the conditions in which they can be accessed (duplication, geographical location) and their associated cost. Thus, for a given requirement, a user could decide which cloud services (from one or several cloud providers) to use for retrieving, processing and integrating data according to the type of contracts she can establish with them.

In this context, data integration deals with a matching problem of the user's integration preferences which includes quality constraints and data requirements, and her specific cloud subscription with the SLA's provided by cloud services. Matching SLAs can imply dealing with heterogeneous SLA specifications and SLA-preferences incompatibilities. Moreover, even with possibility of having an unlimited access to cloud resources, the user is limited to the resources and the budget agreed by her cloud subscription. The aim of this paper is tackle these problems and design a data integration solution guided by SLAs in a multi-cloud context. In addition, we present our service-based query rewriting algorithm guided by user preferences and SLAs as proof of concept.

This paper is organized as follows. Section 2 discusses related works. Section 3 describes the scenario and some challenges. Section 4 introduces our data integration approach. Section 5 describes the *Rhone* algorithm and its formalization. Experiments and results are described in the section 6. Finally, section 7 concludes the paper and discusses future works.

2 Related works

In recent years, the cloud have been the most popular deployment environment for data integration [5]. Existing works addressing this issue can be grouped according to two different lines of research: *(i)* data integration and services [6, 9, 14, 15]; and *(ii)* service level agreements (SLA) and data integration [3, 12].

[6] proposed a query rewriting method for achieving RDF data integration. The objective of the approach is to: (i) solve the entity co-reference problem which can lead to ineffective data integration; and (ii) exploit ontology alignments with a particular interest in data manipulation. [9] introduced a system (called SODIM) which combines data integration, service-oriented architecture and distributed processing. The novelty of these approaches is that they perform data integration in service oriented contexts, particularly considering data services. They also take into consideration the requirement of computing resources for integrating data. Thus, they exploit parallel settings for implementation costly data integration processes.

A major concern when integrating data from different sources (services) is privacy that can be associated to the conditions in which integrated data collections are built and shared. [15] focused on data privacy based on a privacy preserving repository in order to integrate data. Based on users' integration re-

quirements, the repository supports the retrieval and integration of data across different services. [14] proposes an inter-cloud data integration system that considers a trade-off between users' privacy requirements and the cost for protecting and processing data. According to the users' privacy requirements, the query plan in the cloud repository creates the users' query. This query is subdivided into sub-queries that can be executed in service providers or on a cloud repository. Each option has its own privacy and processing costs. Thus, the query plan executor decides the best location to execute the sub-query to meet privacy and cost constraints.

Service level agreement (SLA) contracts have been widely adopted in the context of cloud computing. Research contributions mainly concern (i) SLA negotiation phase (step in which the contracts are established between customers and providers) and (ii) monitoring and allocation of cloud resources to detect and avoid SLA violations. [12] proposed a data integration model guided by SLAs in a grid environment. Their work use SLAs to define database resources. Then, resources can be evaluated (in terms of processing cost, amount of data and price of using the grid) and selected to the integration. A matching algorithm is proposed to produce query plans. The most appropriated solutions based on the QoS are selected as final results. Apart from our previous work [3], to the best of our knowledge, there is no evidence of researches on SLA applied to data integration in a (multi-)cloud context.

The main aspect in a data integration solution is the query rewriting. In the database domain, the query rewriting problem using views have been widely discussed [10, 11, 8, 13]. Similarly, data integration can be seen in the service-oriented domain as a service composition problem in which given a query the objective is to lookup and compose data services that can contribute to produce a result.

Generally, data integration solutions on the service-oriented domain deal with query rewriting problems. [2] proposed a query rewriting approach which processes queries on data provider services. [4] introduced a service composition framework to answer preference queries. Two algorithms inspired on [2] are presented to rank the best rewritings based on previously computed scores. [1] extended [7] and presented an refinement algorithm based on *MiniCon* that produces and order rewritings according to user preferences and scores used to rank services that should be previously define by the user. In general, these approaches shares the same performance problem as the traditional database algorithms. Furthermore, they do not take into consideration user's integration requirements what can lead to produce rewritings that are not satisfactory to the user in terms of quality requirements and cost.

3 Scenario

Let us assume the following scenario in the medical domain. Users are able to retrieve information about (i) patients that were infected by a disease; (ii) regions most affected by a disease in Europe; (iii) patients' personal information; and

(iv) patients' DNA information. There are four services to perform these actions: **S1** retrieves the list of infected patients given a disease name; **S2** it returns the list of cities most affected given a disease name; **S3** retrieves patient' personal information given the patient' id; and **S4** retrieves patient' DNA information given the patient's id.

Doctor Marcel would like to study the type of people suffering of a particular disease. For such reason, he needs to query the patients' personal information and patients' DNA information from patients that were infected by flu. Presuming that Marcel has at his disposal access to a cloud including the services previously described. To achieve his needs, Marcel could compose the data services as follows: (i) he invokes service **S1** with the disease information then he gets the set of people infected by flu; (ii) then he invokes service **S3** with the obtained patients in order to retrieve their personal information; just after (iii) Doctor Marcel invokes service **S4** with the obtained patients to retrieve their DNA information.

In this scenario, depending on the amount of services available to Marcel, many service compositions could be produced to answer Marcel's query. As mentioned in the section 2, some authors have tackled this issue. However, the proposals share the same problems: (i) producing service compositions when a considerable amount of services are available is extremely expensive (in terms of performance); (ii) not always the quality of the rewritings produced is enough for meeting user's requirements; and (iii) executing rewritings that are not satisfactory for the user, generates an extra cost (in term of economic price).

4 SLA guided data integration

Motivated by the problems highlighted in the previous section, we propose a new vision of data integration.

Assuming the medical scenario and Marcel's interest in the type of people suffering of a particular disease as before. However, in this new context, he is also capable to express his preferences while integrating services. For instance, he needs to query the patients' personal information and patients' DNA information from patients that were infected by flu, using services with availability higher than 98%, price per call less than 0.2$ and total cost less than 2$. Marcel has at his disposal a set of services **S1**, **S2**, **S3** and **S4** geographically disposed on different cloud provides (configuring a multi-cloud environment). To achieve his needs, Marcel can use the data services as before invoking in sequence the service **S1**, **S3** and **S4**. However, in this new context, the service selection and rewriting process should meet the user' requirements.

Thus, given a user query, a set of user preferences associated to it, cloud providers and services, our SLA guided data integration process can be divided in four steps.

SLA derivation. This step creates an *integrated SLA* that includes a set of measures corresponding to the user preferences. The *integrated SLA* guides the query

evaluation, and the way results are computed and delivered.
Filtering data services. The *integrated SLA* is used (i) to filter previous SLA derived for a similar request in order to reuse previous results; or (ii) to filter possible data services that can be used for answering the query.
Query rewriting. Given a set of data services that can potentially provide data for integrating the query result, a set of service compositions is generated according to the *integrated SLA* and the agreed SLA of each data service.
Integrating a query result. The service compositions are executed in one or several clouds where the user has a subscription. The execution cost of service compositions must fulfill the *integrated SLA* (that expresses user requirements). Here, the clouds resources needed to execute the composition and how to use them is decided taking in consideration the economic cost determined by the data to be transferred, the number of external calls to services, data storage and delivery cost.

Although *the SLA derivation* is the big challenge while dealing with SLAs and particularly for adding quality dimensions to data integration, the focus in this paper is our query rewriting algorithm which deals with user preferences and SLAs exported by different cloud providers and data services. Here, we are assuming that there is a mechanism responsible to extract the services' quality aspects from SLA, and to provide this information as input to the algorithm. The figure 1 illustrates the structure of SLA and its measures that are used inside our approach.

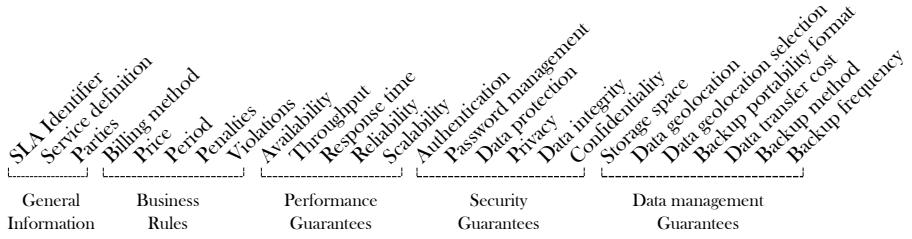

Fig. 1: Cloud SLA

5 Rhone service-based query rewriting algorithm

This section describes the service-based query rewriting algorithm, called *Rhone*.

Given a set of *abstract services*, a set of *concrete services*, a *user query* and a set of *user quality preferences*, the *Rhone* derives a set of service compositions. These service compositions answer the query and that fulfill the quality preferences regarding the context of data service deployment.

The algorithm consists in four macro-steps: (*i*) select services; (*ii*) create variable mappings from services to the query; and (*iii*) combine the services and (*iv*) produce rewriting that matches with the query.

The input for the *Rhone* algorithm is: (1) a query, and (2) a list of concrete services.

Definition 1 (query). *A query Q is defined as a set of abstract services, a set of constraints, and a set of user preferences in accordance with the grammar:*

$$Q(\overline{I}_h;\overline{O}_h) := A_1(\overline{I}_{1l};\overline{O}_{1l}), A_2(\overline{I}_{2l};\overline{O}_{2l}), .., A_n(\overline{I}_{nl};\overline{O}_{nl}), C_1, C_2, .., C_m[P_1, P_2, .., P_k]$$

The left-hand of the definition is called the head of the query; and the right-hand is called the body. $\overline{I}$ *and* $\overline{O}$ *are a set of comma-separated input and output parameters, respectively. Parameters can be of two types: head variables and local variables. Head variables are parameters appearing in the head of the query. They also appear in the body of the query. Local variables are parameters appearing only in the body of the query. The sets of input and output parameters tagged with a subscript h or l refer to head or local parameters, respectively. Two rules are applied to those parameters: the union and the intersection. For instance, the union of head and local input variables builds* $\overline{I}$ *such as* $\overline{I} = \overline{I}_h \cup \{\overline{I}_{1l}, .., \overline{I}_{nl}\}$*; the intersection of head and local input variables is never empty such as* $\{\overline{I}_h \cap \overline{I}_{1l} \cap \overline{I}_{2l}, .., \cap \overline{I}_{nl}\} \neq \emptyset$*. The same example can be used to output variables.*

Abstract services $(A_1, A_2, .., A_n)$ *describe a set of basic service operations.* $C_1, C_2, .., C_m$ *are constraints over the input and/or output parameters. These constraints are used while querying the databases. The user preferences (over the services) are specified in* $P_1, P_2, .., P_k$*.* C_i *and* P_j *are in the form* $x \otimes c$*, where* x *is a identifier;* c *is a constant; and* $\otimes \in \{\geq, \leq, =, \neq, <, >\}$*.*

User preferences can be of two types, single and composed. Single preferences are associated directly to each service involved in the composition. Composed preferences are linked to the entire composition. They are defined in terms of single preferences. For instance, the total response time is a composed preference obtained by adding the response time of each service involved in the composition.

Example 1. Let us suppose a query specification based on the scenario (section 3). The decorations ? and ! differentiate input and output parameters, respectively.

$$Q(dis?; dna!, info!) := GetPatients(dis?; p!), GetDNA(p?; dna!), GetInfo(p?; info!)$$
$$d = \text{“}flu''[availability > 98\%,\ price\ per\ call < 0.2\$,\ total\ cost < 2\$]$$

The user provides a disease name and expects to retrieve the DNA and personal information of patients infected by the given disease. The query execution plan begins by retrieving infected patients ($GetPatients$). This operation returns patients' ids p. The abstract services $GetDNA$ and $GetInfo$ use patient ids to return their DNA and personal information (dna and $info$). The query contains a constraint d (disease name) equal to flu, and three user preferences $availability$ higher than 98 percent, *price per call* less than 2 cents, and *total cost* less than 2 dollars.

Definition 2 (concrete services). *A concrete service (S) is defined as a set of abstract services, and by its quality measures according to the grammar:*

$$S(\overline{I}_h; \overline{O}_h) := A_1(\overline{I}_{1l}; \overline{O}_{1l}), A_2(\overline{I}_{2l}; \overline{O}_{2l}), .., A_f(\overline{I}_{fl}; \overline{O}_{fl})[M_1, M_2, .., M_g]$$

A concrete service definition is similar to the query definition, excepting it does not have constraints. Parameters type and rules are applied in the same way. Concrete services are also defined in terms of abstract services ($A_1, A_2, .., A_n$). They contain a set of service's quality aspects (quality measures) $M_1, M_2, .., M_g$. These measures are associated to the concrete service itself and reflect the quality aspects guaranteed by the service. These aspects and penalties for its violation are agreed between the service and the provider in the service level agreement (SLA). M_i is in the form $x \otimes c$, where x is a special class of identifiers associated to the services; c is a constant; and $\otimes \in \{\geq, \leq, =, \neq, <, >\}$.

In this algorithm, we are assuming this inputs come from a previous phase in our approach. This phase allows (i) to extract the service's quality measures from SLAs; and (ii) to generate the expected input data according to the grammar.

Example 2. Assuming the query Q specified in the *Example 1*, five concrete services (that could be composed to answer it) are exemplified below.

$S1(d?; p!) := GetPatients(d?; p!)[availability > 99\%,\ price\ per\ call = 0.1\$]$
$S2(d?; p!) := GetPatients(d?; p!)[availability > 97\%,\ price\ per\ call = 0.2\$]$
$S3(p?; dna!) := GetDNA(d?; dna!)[availability > 98\%,\ price\ per\ call = 0.1\$]$
$S4(p?; info!) := GetInfo(d?; dna!)[availability > 98\%,\ price\ per\ call = 0.1\$]$
$S5(d?; dna!) := GetPatients(d?; p!), GetDNA(p?; dna!)[availability > 98\%,\ price\ per\ call = 0.1\$]$

$S1$, $S2$, $S3$, $S4$ and $S5$ are different concrete services defined in terms of abstract services or composition of abstract services (*i.e.* $S5$). Each concrete service is tagged with its own quality measures. $S1$ and $S2$ retrieve infected patients, but they differ on the quality measures. $S3$ returns DNA information from a given patient. $S4$ retrieves personal information from patients. Finally, $S5$ covers two abstract services. It returns infected patients and their DNA information.

5.1 Overview on the algorithm

The main function of the *Rhone* is described in the algorithm 1. The input data for this function is a query, which includes a set of user preferences, and a set of concrete services. The result is a set of rewriting of the query in terms of concrete services, fulfilling the user preferences.

In the first step, the algorithm looks for concrete services that can be matched with the query (line 2), resulting in a set of candidate concrete services. For this set of services, the Rhone tries to create *concrete services description* (CSD) for each service (line 3). A CSD is a structure that maps a concrete service to the query, or part of it. The result of this step is a list of CSDs. Given all produced CSDs (line 4), they are combined among each other to generate lists of CSD

Algorithm 1 - RHONE

Input: A query Q, a set of user preferences, and a set of concrete services $\mathcal{S}$.
Output: A set of rewritings R that matches with the query and fulfill the user preferences.

1: **function** $rhone(Q, \mathcal{S})$
2: $\mathcal{L}_{\mathcal{S}} \leftarrow SelectCandidateServices(Q, \mathcal{S})$
3: $\mathcal{L}_{\mathcal{CSD}} \leftarrow CreateCSDs(Q, \mathcal{L}_{\mathcal{S}})$
4: $I \leftarrow CombineCSDs(\mathcal{L}_{\mathcal{CSD}})$
5: $R \leftarrow ProduceRewritings(Q, I)$
6: **return** R
7: **end function**

combinations, in each element represents a possible rewriting. Finally, given the list of combinations, the *Rhone* identifies the ones matching with the query and fulfilling the user preferences (line 5). In the next sections, each phase of the algorithm is described in detail.

5.2 Selecting services

One contribution of our approach concerns the services selection process. Services are selected based on the user preferences and on the services' quality aspects collected from service level agreements. While selecting services, the algorithm deals with three matching problems: *measures* matching, *abstract service* matching and *concrete service* matching.

Definition 3 (measures matching). *Given a user preference P_i and a service's quality measure Q_j, a matching between them can be made if: (i) the identifier c_i in P_i has the same name of c_j in Q_j; and (ii) the evaluation of Q_j, denoted $eval(Q_j)$, must satisfy the evaluation of P_i ($eval(P_i)$). In other words, $eval(Q_j) \subset eval(P_i)$.*

Definition 4 (abstract service matching). *Given two abstract services A_i and A_j, a match between abstract services occurs when an abstract service A_i can be matched to A_j, denoted $A_i \equiv A_j$, according to the following conditions: (i) A_i and A_j must have the same abstract function name; (ii) the number of input variables of A_i, denoted $vars_{input}(A_i)$, is equal or higher than the number of input variables of A_j ($vars_{input}(A_j)$); and (iii) the number of output variables of A_i, denoted $vars_{output}(A_i)$, is equal or higher than the number of output variables of A_j ($vars_{output}(A_j)$).*

Definition 5 (concrete service matching). *A concrete service S can be matched with the query Q according to the following conditions: (i) $\forall A_i$ s. t.$\{ A_i \in S\}$, $\exists A_j$ s. t.$\{ A_j \in Q\}$, where $A_i \equiv A_j$. For all abstract services A_i in S, there is one abstract service A_j in Q that satisfies the abstract service matching problem (Definition 4); and (ii) for all single preferences P_i in Q, there is one service quality measure Q_i in S that satisfies the measures matching problem (Definition 3).*

The process of selecting candidate concrete services is described in the algorithm 2. Given the query and a set of concrete services, the algorithm looks for concrete services that can be used in the rewriting process. While iterating all concrete services in the list $\mathcal{S}$ (line 3), firstly, each service is checked to analyze if all its quality measures satisfies the user preferences in Q (line 4). If it satisfies, each abstract service in S_i is checked to confirm if it matches or not with the query (lines 6-11). Once the service satisfies all the matching problems, a set of candidate concrete services is produced (line 12-13). The result is a list of *candidate concrete services* $\mathcal{L}_\mathcal{S}$ which probably can be used in the rewriting process (line 17).

Algorithm 2 - Select candidate services

Input: A query Q and a set of concrete services $\mathcal{S}$.
Output: A set of candidate concrete services $\mathcal{L}_\mathcal{S}$ that can be used in the rewriting process and fulfill the user preferences.

```
1: function SelectCandidateServices(Q, S)
2: L_S ← ∅
3: for all S_i in S do
4:    if SatisfyQualityMeasures(Q, S_i) then
5:       b ← true
6:       for all A_j in S_i do
7:          if Q.notContains(A_i) then
8:             b ← false
9:             break
10:         end if
11:      end for
12:      if b = true then
13:         L_S ← L_S ∪ {S_i}
14:      end if
15:   end if
16: end for
17: return L_S
18: end function
```

Example 3. The *Rhone* iterates in the concrete service list looking for services satisfying the matching problems. Taking into account the query and concrete services specified in the *Examples 1* and *2*: $S1$, $S3$, $S4$ and $S5$ are selected as candidate concrete service once they satisfy all matching problems. However, $S2$ is not select once it measures violate the user preference *availability* which is higher than 97% and the user expects higher than 98%.

5.3 Candidate service description

After producing the set of candidate concrete services, the next step creates candidate service descriptions (CSDs). A CSD maps abstract services and variables of a concrete service into abstract services and variables of the query.

Definition 6 (candidate service description). *A CSD is represented by an n-tuple:*

$$\langle S, h, \varphi, G, P \rangle$$

where S is a concrete service. h are mappings between variables in the head of S to variables in the body of S. φ are mapping between variables in the concrete service to variables in the query. G is a set of abstract services covered by S. P is a set quality measures associated to the service S.

A CSD is created according to 4 rules: (*i*) for all head variables in a concrete service, the mapping h from the head to the body definition must exist; (*ii*) Head variables in concrete services can be mapped to head or local variables in the query; (*iii*) Local variables in concrete services can be mapped to head variables in the query; and (*iv*) Local variables in concrete services can be mapped to local variables in the query if and only if the concrete service covers all abstract services in the query that depend on this variable. The relation "depends" means that this an output local variable is used as input in another abstract service.

The algorithm 3 describes the creation of CSDs. Given the query Q and a list of candidate concrete services $\mathcal{L}_{\mathcal{S}}$, a list of CSDs $\mathcal{L}_{\mathcal{CSD}}$ is produced. The algorithm iterates on each service in $\mathcal{L}_{\mathcal{S}}$ (line 3), verifying if the mappings rules are being satisfied (line 4). For the ones which satisfies the mapping rules, a fresh copy of the abstract services in the concrete service is done in G(lines 7-9) and a copy of the service quality measures in done in P (lines 10-12). Then, a CSD is created (line 13), and added to the final list of CSDs $\mathcal{L}_{\mathcal{CSD}}$ (line 14). The result of this phase is a list of CSDs that can be used to build rewriting of the query (line 17).

Example 4. Given the candidate concrete services selected in the *Example 3*. The algorithm builds CSDs to concrete services satisfying the mapping rules. $S1$, $S3$ and $S4$ satisfy all mapping rules. Consequently, CSDs for them are created. For instance, CSD_1 is produced to $S1$ as follows: $\langle S1,\ h = \{d \rightarrow d,\ p \rightarrow p\},\ \varphi = \{d \rightarrow dis,\ p \rightarrow p\},\ G = \{GetPatients\},\ P = \{availability > 99\%,\ price\ per\ call = 0.1\$\}\rangle$. However, a CSD for $S5$ is not build because it violates the rule for local variables. It contains a local variable (p) mapped to a local variable in the query. Consequently, $S5$ must cover all abstract services in the query depending on this variable, but the abstract service $GetInfo$ is not covered.

5.4 Combining and producing rewritings

Given the list of CSDs $\mathcal{L}_{\mathcal{CSD}}$ produced, the *Rhone* produces all possible combinations of its elements. Building combinations I (Algorithm 1, line 4) deals with

Algorithm 3 - Create candidate service descriptions (CSDs)

Input: A query Q and a set of candidate concrete services $\mathcal{L}_{\mathcal{S}}$.
Output: A set of candidate service descriptions (CSDs) $\mathcal{L}_{\mathcal{CSD}}$ that contains mappings from candidate concrete service to the query.

```
1: function CreateCSDs(Q, L_S)
2: L_CSD ← ∅
3: for all S_i in L_S do
4:    if There are mappings h and φ from S_i to Q then
5:       G ← ∅
6:       P ← ∅
7:       for all A_j in S_i do
8:          G ← G ∪ {A_j}
9:       end for
10:      for all M_k in S_i do
11:         P ← P ∪ {M_k}
12:      end for
13:      CSD := ⟨S_i, h, φ, G, P⟩
14:      L_CSD ← L_CSD ∪ {CSD}
15:   end if
16: end for
17: return L_CSD
18: end function
```

a NP hard complexity problem. The effort to process combinations increases while the number of CSDs and abstract services in the query increases.

The last step identifies rewritings matching with the query and fulfilling the user preferences (Algorithm 4). The set of rewritings R is initialized empty (line 2). Another contribution in our algorithm concerns the aggregation functions $\mathcal{T}_{\text{init}}[\![\ \mathcal{A}gg(Q)\ ]\!]$, $\mathcal{T}_{\text{cond}}[\![\ \mathcal{A}gg(Q)\ ]\!]$ and $\mathcal{T}_{\text{inc}}[\![\ \mathcal{A}gg(Q)\ ]\!]$. They are responsible to initialize (line 3), check conditions (line 5) and increment (line 8) composed preferences defined by the user. This means for each element in the CSD list p the value of a composed measure is computed and incremented. Rewritings are produced while the user preferences are respected.

The *Rhone* algorithm verifies if a given CSD list p is a rewriting of the original query (line 6). The algorithm 5 describes this process in detail. Given the CSD list p (line 2), the function return *true* if it is a rewriting of the query. p is a rewriting if it satisfies two conditions: (i) the number of abstract services resulting from the union of all CSDs in p must be equals to the number of abstract services in the query; and (ii) the intersection of all abstract services in each CSD on p must be empty. It means that is forbidden to have abstract services replicated among the set p.

Example 5. Let us consider the CSDs CSD_1, CSD_3 and CSD_5 produced in the *Example 4* refers to the concrete services $S1$, $S3$ and $S4$, respectively. The *Rhone* produces combinations as follows:

$p_1 = \{CSD_1\}$

Algorithm 4 - Producing rewritings

Input: A query Q and a list of lists of CSDs I.
Output: A set of rewritings R that matches with the query and fulfill the user preferences.

function *ProduceRewritings*(Q, I)
$R \leftarrow \emptyset$
$\mathcal{T}_{\text{init}}[\![\ \mathcal{A}gg(Q)\ ]\!]$
$p \leftarrow I.next()$
while $p \neq \emptyset$ **and** $\mathcal{T}_{\text{cond}}[\![\ \mathcal{A}gg(Q)\ ]\!]$ **do**
  if *isRewriting*(Q, p) **then**
    $R \leftarrow R \cup Rewriting(p)$
    $\mathcal{T}_{\text{inc}}[\![\ \mathcal{A}gg(Q)\ ]\!]$
  end if
  $p \leftarrow I.next()$
end while
return R
end function

Algorithm 5 - Validating a combination of CSDs

Input: A query Q and a set of candidate services descriptions p.
Output: A boolean value. *True*, if the set p is a rewriting of the query. *False*, otherwise.

function *isRewriting*(Q, p)
let $p = \{CSD_1, CSD_2, ..., CSD_k\}$
if (a) The number of elements in the union $CSD_1.G_1 \cup CSD_2.G_2, ..., \cup CSD_k.G_k$ is equal to the number of abstract services in Q
(b) The intersection $CSD_1.G_1 \cap CSD_2.G_2, ..., \cap CSD_k.G_k$ is empty **then**
  return $true$
end if
return $false$
end function

$p_2 = \{CSD_1, CSD_3\}$
$p_3 = \{CSD_1, CSD_3, CSD_4\}$

Given the combinations, the *Rhone* checks if each one of them is a valid rewriting of the original query.

- p_1 and p_2 are not valid rewritings; their number of abstract services do not match with the number of abstract services in the query.
- p_3 is a valid rewriting; the number of abstract services matches and there is no repeated abstract service.

6 Evaluation

This section describes the experiments performed as proof of concept to the algorithm. The Rhone prototype is implemented in Java. It includes 15 java

classes in which 14 of them model the basic concepts (*query*, *abstract services*, *concrete services*, etc), and 1 responsible to implement the core of the algorithm.

Currently, our approach runs in a controlled environment. Different experiments were produced to analyze the algorithm's behavior. We will present two experiments: *experiment 1* and *experiment 2*. The service registry used has 100 concrete services. In each experiment, there are a set of tests in which the number of concrete services varies from 5 until to reach 100.

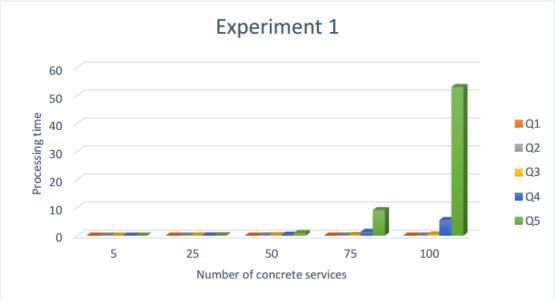

Fig. 2: Query rewriting evaluation.

In the *experiment 1* (figure 2), there are five different queries that differ on the quantity of abstract services (increasing from 2 to 6). Analyzing the first experiment, it is easily to identify that the algorithm shares the same problem as existing query rewriting approaches using views: increasing the processing time when the size of the query and the number of concrete services increase.

The *experiment 2* (figure 3) presents the results while testing the algorithm in the presence of user preferences and services' quality aspects extracted from SLAs. The difference between *Test 1* and *Test 2* concerns the way services are selected and the query is rewritten. Once *Test 1* do not consider quality measures as any other existing query rewriting approach, *Test 2* uses the user preferences statements and services' quality aspects to guide the service selection and query rewriting. Both include queries with six abstract services and quality requirements concerning availability, response time, price per call and integration cost (total cost). The figure 3 shows our results.

The results while considering user preferences and SLAs are promisingly. The *Rhone* increases performance reducing rewriting number (around 50 percent) which allows to go straightforward to the rewriting solutions that are satisfactory avoiding any further backtrack and thus reducing successful integration time. Moreover, once the services selection and service composition (rewritings) process are fully guided by the user requirements and SLA, the algorithm avoid producing and executing composition that are not interest for the user. In this sense, the reduces the integration economic cost while delivering the expected results.

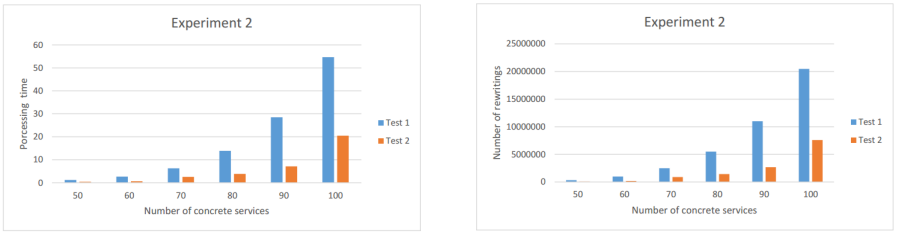

Fig. 3: Results concerning processing time (left-side) and rewritings number (right-side).

7 Final Remarks and Future Works

This work proposes a query rewriting algorithm for data integration quality named *Rhone*. Given a query, user preferences and a list of concrete services as input, the algorithm derives rewritings in terms of concrete services that matches with the query and fulfill the user preferences. The formalization and experiments are presented. The results show that the *Rhone* reduces the rewriting number and processing time while considering user preferences and services' quality aspects extracted from SLAs to guide the service selection and rewriting. We are currently performing improvements in the implementation and setting up a multi-cloud simulation in in order to evaluate the performance of the *Rhone* in such context.

References

1. Ba, C., Costa, U., H. Ferrari, M., Ferre, R., A. Musicante, M., Peralta, V., Robert, S.: Preference-driven refinement of service compositions. In: Int. Conf. on Cloud Computing and Services Science, 2014. Proceedings of CLOSER 2014 (2014)
2. Barhamgi, M., Benslimane, D., Medjahed, B.: A query rewriting approach for web service composition. Services Computing, IEEE Transactions on 3(3), 206–222 (July 2010)
3. Bennani, N., Ghedira-Guegan, C., Musicante, M., Vargas-Solar, G.: Sla-guided data integration on cloud environments. In: 2014 IEEE 7th International Conference on Cloud Computing (CLOUD). pp. 934–935 (June 2014)
4. Benouaret, K., Benslimane, D., Hadjali, A., Barhamgi, M.: FuDoCS: A Web Service Composition System Based on Fuzzy Dominance for Preference Query Answering (Sep 2011), http://liris.cnrs.fr/publis/?id=5120, vLDB - 37th International Conference on Very Large Data Bases - Demo Paper
5. Carvalho, D.A.S., Souza Neto, P.A., Vargas-Solar, G., Bennani, N., Ghedira, C.: Database and Expert Systems Applications: 26th International Conference, DEXA 2015, Valencia, Spain, September 1-4, 2015, Proceedings, Part II, chap. Can Data Integration Quality Be Enhanced on Multi-cloud Using SLA?, pp. 145–152. Springer International Publishing (2015)
6. Correndo, G., Salvadores, M., Millard, I., Glaser, H., Shadbolt, N.: SPARQL query rewriting for implementing data integration over linked

data. In: Proceedings of the 1st International Workshop on Data Semantics - DataSem '10. p. 1. ACM Press, New York, New York, USA (Mar 2010), http://dl.acm.org/citation.cfm?id=1754239.1754244 http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.177.8935
7. Costa, U., Ferrari, M., Musicante, M., Robert, S.: Automatic refinement of service compositions. In: Daniel, F., Dolog, P., Li, Q. (eds.) Web Engineering, Lecture Notes in Computer Science, vol. 7977, pp. 400–407. Springer Berlin Heidelberg (2013)
8. Duschka, O.M., Genesereth, M.R.: Answering recursive queries using views. In: Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. pp. 109–116. PODS '97, ACM, New York, NY, USA (1997), http://doi.acm.org/10.1145/263661.263674
9. ElSheikh, G., ElNainay, M.Y., ElShehaby, S., Abougabal, M.S.: SODIM: Service Oriented Data Integration based on MapReduce. Alexandria Engineering Journal 52(3), 313–318 (Sep 2013), http://www.sciencedirect.com/science/article/pii/S111001681300029X
10. Halevy, A.Y.: Answering queries using views: A survey. The VLDB Journal 10(4), 270–294 (Dec 2001), http://dx.doi.org/10.1007/s007780100054
11. Levy, A.Y., Rajaraman, A., Ordille, J.J.: Querying heterogeneous information sources using source descriptions. In: Proceedings of the 22th International Conference on Very Large Data Bases. pp. 251–262. VLDB '96, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996), http://dl.acm.org/citation.cfm?id=645922.673469
12. Nie, T., Wang, G., Shen, D., Li, M., Yu, G.: Sla-based data integration on database grids. In: Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International. vol. 2, pp. 613–618 (July 2007)
13. Pottinger, R., Halevy, A.: Minicon: A scalable algorithm for answering queries using views. The VLDB Journal 10(2-3), 182–198 (Sep 2001), http://dl.acm.org/citation.cfm?id=767141.767146
14. Tian, Y., Song, B., Park, J., nam Huh, E.: Inter-cloud data integration system considering privacy and cost. In: Pan, J.S., Chen, S.M., Nguyen, N.T. (eds.) ICCCI (1). Lecture Notes in Computer Science, vol. 6421, pp. 195–204. Springer (2010)
15. Yau, S.S., Yin, Y.: A privacy preserving repository for data integration across data sharing services. IEEE T. Services Computing 1(3), 130–140 (2008)