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. Nowadays, data provision is mostly done by data services. Data integration can be seen as composition of data services and data processing services that can deal with to integrate data collections. With the advent of cloud, producing service compositions is computationally costly. Furthermore, executing them can require a considerable amount of memory, storage and computing resources that can be provided by clouds. Our research focuses on how to enhance the results on the increase of cost on data integration in the new context of cloud. To do so, we present in this paper our original data integration approach which takes into account user's integration requirements while producing and delivering the results. The service selection and service composition are guided by the service level agreement - SLA exported by different services (from one or more clouds) and used by our matching algorithm (called *Rhone*) that addresses the query rewriting for data integration presented here as proof of concept.

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 under which these services are delivered can defined in contracts using 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 he/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 he/she established 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 to the budget agreed by his/her cloud subscription. The aim of this paper is to tackle these problems and to present a data integration solution guided by SLAs which enhances the quality on the results integration 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 introduces our data integration approach considering a running scenario and the data integration challenges. Section 4 describes the *Rhone* algorithm and its formalization. Experiments and results are described in the section 5. Finally, section 6 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].

In [6], the authors propose a query rewriting method for achieving RDF data integration. The objective of the approach is: (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] introduces a system (called SODIM) which combine 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 use parallel settings for implementing 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] focuses on data privacy in order to integrate data. Based on users' integration requirements, 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 execution 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. These approaches are mostly interested in privacy and performance issues forgetting other users' integration requirements.

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] proposes a data integration model guided by SLAs in a grid environment. Their work uses 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 in order to guide and enhance the quality on 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] proposes a query rewriting approach which processes queries on data provider services. [4] introduces 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] extends [7] and presents 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 share the same performance problem as the traditional database algorithms. Furthermore, they do not take into consideration user's integration requirements which can lead to produce rewritings that are not satisfactory to the user in terms of quality requirements and cost.

3 SLA guided data integration in cloud evironments

With the advent of cloud and multi-clouds a new vision of data integration is to be considered. In this vision (i) *data services* provide access and allow data retrieval; (ii) *data processing services* retrieve and process data; (iii) *data services* and *data processing services* can be deployed in different *cloud providers* geographically distributed; (iv) *data services*, *data processing services* and the *cloud* itself export their SLA defining the services they provide and the quality conditions under which they are delivered. Different levels of SLA are considered: contracts agreed between *data services* or *data processing services* and *cloud providers*, called *cloud SLA* (SLA_C); and contracts agreed between users and *data services* or *data processing services*, called *service SLA* (SLA_S). Therefore, given a user query, his/her integration quality requirements and his/her cloud subscription, the query is rewritten in terms of cloud services (*data services* and *data processing services*) composition that fulfill the integration requirements and deliver the expected results to the user.

To better understand our vision and challenges, let us consider the following medical scenario in which users are able to retrieve and integrate data concerning (i) *patients that were infected by a disease;* (ii) *regions most affected by a disease*; (iii) *patients' personal information*; and (iv) *patients' DNA information.* For instance, doctor *Marcel* would like to study the type of people suffering of *flu* in the Europe. To perform this, he has at his disposal a set of cloud services delivered by different *cloud providers.* Each cloud service and *cloud provider* describe their quality guarantees in SLA contracts. Thereby, to reach his needs, he wants to query the personal and DNA information from patients infected by *flu*, using cloud services with availability higher than 98%, price per call less than 0.2$ and integration total cost less than 5$. This scenario introduces new challenges to data integration that have to be addressed, such as:

- Data integration tasks can require a huge amount of resources and processing time given the large quantity of *data services* and *data processing services*;
- The cloud model comes with the possibility of an on-demand and pay-per-use access to resources. However, cloud consumers have a restricted to the resources according to the contract they have with the cloud, and to the budget they are ready to pay while integrating data.
- Part of the rewritings produced and executed to the user *query* could not be in accordance with her quality requirements. In addition, generating these unsatisfactory rewritings imply increasing the processing time and integration cost. Consequently, the user may become dissatisfied with the results.
- Performing data integration in this scenario implies a matching problem of users' requirements with different services' guarantees specified in SLAs. Considering the multi-cloud environment, we can face different cases of SLA incompatibilities given the different semantics and structure of SLAs exported by the different cloud services and *cloud providers.*
- Executing data integration tasks is computationally costly in terms of processing time and economic cost. Therefore, it is crucial to reuse previous

integration results in order to save time and money while satisfying the user.

Motivated by the challenges discussed above, the figure 1 illustrates our SLA-based data integration workflow. Given a user query, a set of user preferences associated to it, cloud providers and cloud services, the workflow can be divided in four steps:

Fig. 1: SLA-based data integration workflow

SLA derivation. In this step, we compute what we call a *derived SLA* that matches user' integration requirements (including quality constraints and data requirements) with the SLA's provided by *cloud services*, given a specific user cloud subscription. The user may have general *preferences* depending on the context he/she wants to integrate his/her data such as economic cost, bandwidth limit, free services, and storage and processing limits. The *SLA derivation* is the big challenge while dealing with SLAs and particularly for adding quality dimensions to data integration. Furthermore, the *derived SLA* guides the query evaluation, and the way results are computed and delivered.
Filtering data services. The *derived SLA* is used (i) to filter previous *integration SLA* derived for a similar request in order to reuse results; or (ii) to filter possible *cloud 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 *derived SLA* and the agreed SLA of each *data services*.
Integrating a query result. The service compositions are executed with services from one or several clouds where the user has a subscription. The execution cost of service compositions must fulfill the *derived SLA*. The clouds resources needed by the user 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 2 illustrates the structure of SLA and its measures that are considered in the algorithm we will detail in the next section.

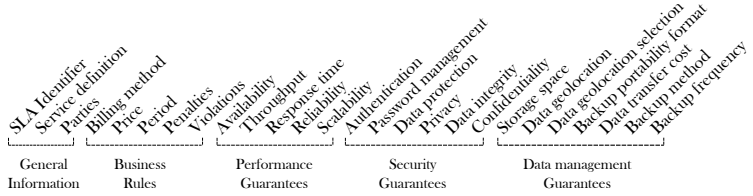


Fig. 2: Cloud SLA model

4 Rhone service-based query rewriting algorithm

This section describes our quality-based query rewriting algorithm called *Rhone*. It is guided by user preferences and services' quality aspects extracted from service level agreements (SLA). Our algorithm has two original aspects: *first*, the user can express his/her quality preferences and associate them to his/her queries; and, *second*, service's quality aspects defined on Service Level Agreements (SLA) guide service selection and the whole rewriting process.

4.1 Preliminaries

The idea behind our algorithm consists in deriving a set of service compositions that fulfill the users' integration preferences concerning the context of data service deployment given a set of *abstract services*, a set of *concrete services*, a user' *query* defined hereafter and a set of user' *integration preferences*.

Definition 1 (*Abstract service*). A *abstract service* describes the small piece of function that can be performed by a cloud service. For instance, retrieve patients, DNA information and personal information. The *abstract service* is defined as $A\ (\overline{I};\ \overline{O})$ where:

- A is a name which identifies the *abstract service*.
- $\overline{I}$ and $\overline{O}$ are a set of comma-separated input and output parameters, respectively.

The table 1 exemplifies *abstract services* concerning our medical scenario described in the section 3. The decorations ? and ! differentiate input and output parameters, respectively.

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

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

The left-hand of the definition is called the *head*; and the right-hand is the *body*. A *concrete service* S includes a set of input $\overline{I}$ and output $\overline{O}$ variables, respectively.

***Abstract service* name**	**Description**
$A1$ ($d\_name?$; $p\_id!$)	Given a disease name d_name, $A1$ returns a list of infected patients' id p_id.
$A2$ ($p\_id?$; $p\_dna!$)	Given a patient id p_id, $A2$ returns her DNA information p_dna.
$A3$ ($p\_id?$; $p\_info!$)	Given a patient id p_id, $A3$ returns her personal information p_info.
$A4$ ($d\_name?$; $regions!$)	Given a disease name d_name, $A4$ returns the most affected region $regions$.

Table 1: List of *abstract services*.

Variables in the *head* are identified by $\overline{I}_h$ and $\overline{O}_h$, and called *head* variables. They appear in the *head* and in the *body* definition. Variables appearing only in the *body* are identified by $\overline{I}_l$ and $\overline{O}_l$, and are called *local* variables. *Head* variables can be accessed and shared among different services. On the other hand, *local* variables can be used only by the service which define them.

Concrete services are defined in terms of *abstract services* $(A_1, A_2, .., A_n)$, and they include a set of service's quality aspects (quality measures) $(M_1, M_2, .., M_g)$. These measures are extracted from the SLA exported by the service S. 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 $\otimes \in \{\geq, \leq, =, \neq, <, >\}$.

Let us consider the following *concrete services* specified using the *abstract services* previously presented to be used as examples to the algorithm:

```
S1(a?;b!) := A1(a?;b!) [availability > 98%, price per call = 0.2$]
S2(a?;b!) := A1(a?;b!) [availability > 98%, price per call = 0.1$]
S3(a?;b!) := A2(a?;b!) [availability > 99%, price per call = 0.1$]
S4(a?;b!) := A1(a?;p!), A2(p?; b!)
                       [availability > 98%, price per call = 0.1$]
S5(a?;b!) := A3(a?;b!) [availability > 98%, price per call = 0.0$]
S6(a?;b!,c!) := A1(a?;p!), A2(p?;b!), A3(p?;c!)
                       [availability > 99%, price per call = 0.2$]
S7(a?;b!) := A4(a?;b!) [availability > 99%, price per call = 0.2$]
```

For instance, $S1$ is written using the *abstract service* $A1$. d and p are *head* variables. *Availability* and *price per call* are identifiers associated to $S1$'s quality measures with an associated constant value extracted from $S1$'s SLA.

Definition 3 (*User query*). A user *query* Q is defined as a set of *abstract services*, a set of *constraints*, and a set of *user integration preferences* in accordance with the following 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 *query* definition is similar to a *concrete service* concerning the variables and *abstract services*. In addition, queries have constraints over the input or output variables $(C_1, C_2, .., C_m)$. Constraints are used while querying the databases (*i.e*

in the *where* clause). The user *integration preferences* over the services or over service compositions are specified in $P_1, P_2, .., P_k$. C_i and P_j are in the form $x \otimes c$, where x is an 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.

Let us suppose a query specification based on the medical scenario (section 3) in which doctor *Marcel* wants to query the personal and DNA information from patients that were infected by *flu*, using services with availability higher than 98%, price per call less than 0.2$ and integration total cost less than 5$. To achieve his needs, the *abstract services* $A1$, $A2$ and $A3$ should be composed as follows.

```
Q(dis?;dna!,info!) := A1(dis?;p!), A2(p?;dna!), A3(p?;info!), d = flu,
             [availability > 98%, price per call < 0.2$, total cost < 5$]
```

The *Marcel's query* plan begins by retrieving infected patients ($A1$). This operation returns patients' ids p. The *abstract services* $A2$ and $A3$ use patient ids to return their DNA and personal information (*dna* and $info$). The *query* contains a constraint *dis* (disease name) equal to flu, and three identifiers define the user' *integration preferences* with their associated constant value: *availability* higher than 98 percent, *price per call* less than 2 cents, and *total cost* less than 5 dollars.

4.2 Overview of the algorithm

The input data for the *Rhone* 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. The main function of the algorithm is divided in four steps:

Select candidate services. The algorithm looks for *concrete services* including *abstract services* that can be matched with the query' *abstract services* considering their name and input/output parameters, resulting in a set of candidate concrete services.

Building CSDs. For each candidate concrete service, the Rhone tries to create *concrete service description* (CSD). 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.

Combining CSDs. Given all CSDs, they are combined among each other to generate lists of CSD combinations in which each element represents a possible rewriting.

Producing rewritings. Finally, given the list of combinations, the *Rhone* identifies the ones matching with the query and fulfilling the user preferences. In the next sections, each phase of the algorithm is described in detail.

4.3 Selecting candidate concrete 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, $eval(Q_j) \subset eval(P_i)$. For instance, *price per call* $< 0.2\$$ is a *user preference* that can be matched with a service' quality measure *price per call* $= 0.1\$$.

Definition 4 (*Abstract service* matching). Given two abstract services A_i and A_j, a matching 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) $\forall\; P_{iQ},\; \exists\; Q_{iS}\; /\; Eval(Q_{iS}) \;=\; true$.

The process of selecting candidate concrete services is described in the algorithm 1. Given the query Q and a set of concrete services $\mathcal{S}$, the algorithm looks for concrete services that can be used in the rewriting process. Each selected service should satisfy the user preferences in Q (line 4), and match with the query (line 7). The services that satisfy all the matching problems are included in a set of candidate concrete services $\mathcal{L}_{\mathcal{S}}$ (line 12-13) which probably can be used in the rewriting process.

Let us consider the *query* and *concrete services* illustrated in the section 4.1. The *Rhone* iterates in the *concrete service* list looking for services satisfying the matching problems. Taking into account the *query* and the *concrete services*, `S2`, `S3`, `S4` and `S5` are selected as candidate concrete services as they satisfy all matching problems. However, `S1`, `S6` and `S7` are not selected once their measures violate the user *integration preference* '*price per call*'. In addition, `S7` is not also selected because it contains a *abstract service* that does not contribute to answer the *query*.

4.4 Candidate service description creation

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.

Algorithm 1 - 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
```

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 2 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. A CSD is created only for candidate concrete services in which the mappings rules are being satisfied (line 4).

Given the candidate concrete services `S2`, `S3`, `S4` and `S5` selected in the previous step. The algorithm builds CSDs to `S2`, `S3` and `S5` once they satisfy all the mapping rules as follows. For instance, CSD_2 is produced to `S2` as fol-

Algorithm 2 - 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
```

lows: $\langle S2,\ h = \{a \rightarrow a,\ b \rightarrow b\},\ \varphi = \{a \rightarrow dis,\ b \rightarrow p\},\ G = \{A1\},\ P = \{availability > 98\%,\ price\ per\ call = 0.1\$\}\rangle$. However, a CSD for `S4` 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, `S4` must cover all abstract services in the query depending on this variable, but the abstract service $A3$ is not covered.

4.5 Combining and producing rewritings step

Given the list of CSDs $\mathcal{L}_{\mathcal{CSD}}$ produced, the *Rhone* produces all possible combinations of its elements. Building combinations I (Algorithm 3) deals with 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 3). 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 p in the CSD list 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 (Algorithm 4, line 6). The algorithm 4 describes this process in detail. Given the CSD list p (line 2), the function return *true* if (i) the number

Algorithm 3 - 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.

1: **function** *ProduceRewritings*(Q, I)
2: $R \leftarrow \emptyset$
3: $\mathcal{T}_{\text{init}}[\![\ \mathcal{A}gg(Q)\ ]\!]$
4: $p \leftarrow I.next()$
5: **while** $p \neq \emptyset$ **and** $\mathcal{T}_{\text{cond}}[\![\ \mathcal{A}gg(Q)\ ]\!]$ **do**
6: **if** *isRewriting*(Q, p) **then**
7: $R \leftarrow R \cup Rewriting(p)$
8: $\mathcal{T}_{\text{inc}}[\![\ \mathcal{A}gg(Q)\ ]\!]$
9: **end if**
10: $p \leftarrow I.next()$
11: **end while**
12: **return** R
13: **end function**

of abstract services resulting from the union of all CSDs in p is equals to the number of abstract services in the query; and (ii) the intersection of all abstract services in each CSD on p is empty. It means that is forbidden to have abstract services replicated among the set p.

Algorithm 4 - 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.

1: **function** *isRewriting*(Q, p)
2: **let** $p = \{CSD_1, CSD_2, ..., CSD_k\}$
3: **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**
4: **return** $true$
5: **end if**
6: **return** $false$
7: **end function**

Let us consider CSD_2, CSD_3 and CSD_5 are CSDs that refer to the concrete services `S2`, `S3` and `S5`, respectively. The *Rhone* produces combinations taking into account the part of the query covered by the service as follows:

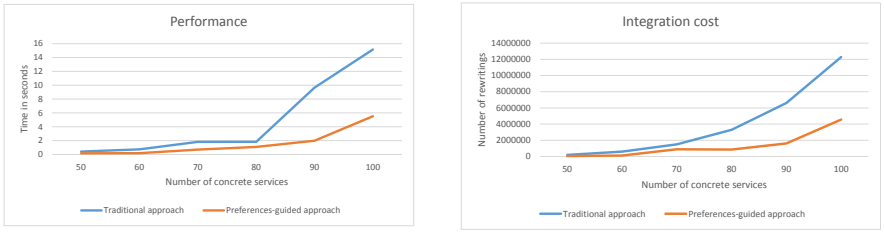
$p_1 = \{CSD_2\}$
$p_2 = \{CSD_2, CSD_3\}$
$p_3 = \{CSD_2, CSD_3, CSD_5\}$

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.

5 Evaluation

Different experiments were produced to analyze the algorithm's behavior. The *Rhone* [6], so far, was evaluated in a local controlled environment simulating a mono-cloud including a service registry of 100 concrete services. Some experiments were produced to analyze the its behavior concerning performance, and quality and cost of the integration. The experiments include two different approaches: (i) the *traditional approach* in which user preferences and SLAs are not considered; and (ii) the *preference-guided approach* (P-GA) which considers the users' integration requirements and SLAs. Figures 3a and 3b summarize our first results.



(a) Performance evaluation.

(b) Integration cost.

Fig. 3: *Rhone* execution evaluation.

The results P-GA are promisingly. Our approach increases performance reducing rewriting number which allows to go straightforward to the rewriting solutions that are satisfactory avoiding any further backtrack and thus reducing successful integration time (Figure 3a). Moreover, using the P-GA to meet the user preferences, the quality of the rewritings produced has been enhanced and the integration economic cost has considerable reduced while delivering the expected results (Figure 3b).

6 Final Remarks and Future Works

Rhone still need to be tested in a large scale case and in a context of parallel multi-tenant to test efficacy. However, the results can show that the *Rhone*

[6] The *Rhone* algorithm is implemented in Java and 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.

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 a multi-cloud simulation 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. 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 Services Computing (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 (2011), 37th International Conference on Very Large Data Bases (VLDB 2011)
5. Carvalho, D.A.S., Souza Neto, P.A., Vargas-Solar, G., Bennani, N., Ghedira, C.: 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. ACM Press, New York, New York, USA (2010)
7. Costa, U., Ferrari, M., Musicante, M., Robert, S.: Automatic refinement of service compositions. In: Web Engineering, Lecture Notes in Computer Science, vol. 7977. Springer Berlin Heidelberg (2013)
8. Duschka, O.M., Genesereth, M.R.: Answering recursive queries using views. In: Proceedings of the Sixteenth Symposium on Principles of Database Systems. pp. 109–116. ACM, NY, USA (1997)
9. ElSheikh, G., ElNainay, M.Y., ElShehaby, S., Abougabal, M.S.: SODIM: Service Oriented Data Integration based on MapReduce. Alexandria Engineering Journal (2013)
10. Halevy, A.Y.: Answering queries using views: A survey. The VLDB Journal 10(4), 270–294 (Dec 2001)
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. San Francisco, CA, USA (1996)
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 pp. 182–198 (2001)
14. Tian, Y., Song, B., Park, J., nam Huh, E.: Inter-cloud data integration system considering privacy and cost. In: ICCCI. 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 (2008)