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

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

[1] Université Jean Moulin Lyon 3, Centre de Recherche Magellan, IAE, France
daniel.carvalho@univ-lyon3.fr
[2] Université Jean Moulin Lyon 3, LIRIS, Centre de Recherche Magellan, IAE, France
chirine.ghedira-guegan@univ-lyon3.fr
[3] LIRIS, INSA-Lyon, France
nadia.bennani@insa-lyon.fr
[4] CNRS-LIG, Grenoble, France
genoveva.vargas@imag .fr
[5] Federal Institute of Education, Sciences and Technology (IFRN), Natal, Brazil
placido.neto@ifrn.edu.br

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

Data integration problem has been studied by many researchers in the database domain. This can be seen in the service-oriented architecture as a service composition issue, such as given a query, the objective is to lookup for data services and compose them in order to achieve a desired result. However, it is computationally costly to find the best service composition to answer a query. Furthermore, executing the composition can lead to retrieve and process data collections that can require important memory, storage and computing resources. The on-demand resource provision imposed by the cloud computing architecture opens challenges to data processing and management.

Computing resources are delivered as services by the cloud. Services are billed and agreed between service providers and service customers under service level agreement (SLA) contracts. Both sides must agree together on quality conditions and penalties under which the service is delivered. Several researches have reported their studies on SLA in different domains [1]. SLA proposals for cloud computing could be divided in two groups: (i) works developing tools and methods to help on SLA negotiation and enforcement phase [14, 12]; and (ii) approaches that monitor contracts and cloud resources in order to detect and avoid SLA violations [9, 11]. To our knowledge, SLA publications have not been yet integrated to data integration in a multi-cloud environment.

Based on these concepts and open challenges, the goal of this work is to present a data integration solution concerning the *Rhone* service-based query rewriting algorithm guided by SLA's. Our work addresses these issues and proposes the algorithm (we called *Rhone*) with two original aspects: (i) the user can express her quality preferences and associate them to her queries; and (ii) service's quality aspects defined on Service Level Agreements (SLA) guide service selection and the whole rewriting process. Yet, to the best of our knowledge, we have not identified any other work that uses SLA to guide the entire data integration solution.

This paper is organized as follows. Section 4 presents some related works. Section 3 contains the running scenario and challenges. 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 Data integration and SLA

Inserindo...

3 Scenario

This section is devoted to describe the motivation scenario and challenges concerning our data integration approach.

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. To perform these actions, four family of services are necessary: family **A** has services which given a disease name, it retrieves the list of infected patients; family **B** has services which given a disease name, it retrieves the list of cities most affected by that disease; family **C** has services which given a patient id, it retrieves patients' personal information; and family **D** has services which given a patient id, it retrieves patients' dna information.

Doctor Marcel would like to study the type of people suffering of a particular disease. For instance, he needs to query the patients' personal information and patients' dna information from the set of patients that were infected by flu. Presuming that Marcel has at his disposal a cloud including a set of services

from the families **A**, **B**, **C** and **D**. To achieve his needs, Marcel can use the data services as follows: (i) he invokes service **S1** (family **A**) with the disease information then he gets the set of people infected by flu; (ii) then he invokes service **S2** (family **C**) with the obtained patients in order to retrieve their personal information; just after (iii) Doctor Marcel invokes service **S3** (family **D**) with the obtained patients to retrieve their dna information. Finally, the query results is integrated and returned.

Depending on the amount of services in each family type, a lot of other service compositions could be done to answer Marcel's query. A large quantity of algorithms for this purpose have been developed, and all of them share the same problems: (1) producing rewritings when a big amount of services are available is extremely expensive; and (2) not always the quality of the rewriting (composition) produced is enough for meeting your needs. Motivated by these problems, our approach proposes a new vision of data integration as follows.

Assuming the same medical scenario and the same families of data services. Let us suppose Marcel would like to study the type of people suffering of a particular disease as before. However, in this new scenario, 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 from the families **A**, **B**, **C** and **D** geographically disposed on different cloud provides (configuring a multi-cloud environment). To achieve his needs, Marcel can use the data services as before invoking one service from the families **A**, **C** and **D** in sequence. However, in this new configuration all the services involved must satisfy the user preferences expressed in the query. The selection and rewriting process is guided by the service level agreements (SLA) exported from different services. The user preferences are matched with the service quality aspects that are defined on its SLAs. This new vision of data integration brings new challenges such as: (i) The amount of services involved in a multi-cloud context is bigger than in a single cloud. Consequently, the number of services that can be used in the rewriting process and the number of rewriting produced is higher. Such environment calls for a better services selection process, *i.e.*, guided by the SLAs and user preferences; (ii) How can the different SLAs associated to services and cloud provider can be integrated with the user preferences? There are different levels of SLAs: the one agreed between services and cloud providers; and the ones agreed between services and users. These SLAs should be integrated; and (iii) How can a previous processed query be reused for a next query?

Here, it is important to highlight that this paper focus on the description and evaluation of the algorithm that rewrites queries in terms of services composition taking into account user preferences and service quality aspects expressed in SLA contracts. We are assuming that the extraction of quality aspects from SLAs is performed in a previous phase of our global data integration solution. In the next section, the Rhone service-based algorithm is described and formalized.

4 Related works

The main aspect in a data integration solution is the query rewriting process executed by a mediator in accordance with the different databases. In this way, algorithms for rewriting queries have been proposed in two domains: (i) on the database domain; and (ii) on the service-oriented domain.

On the database domain, query rewriting approaches using views have been widely discussed [8]. For instance, the *bucket algorithm* [10], *inverse-rules algorithm* [7] and *MiniCon algorithm* [13] have tackled the rewriting problem on the database domain. In addition, [13] has inspired rewriting methods in service-oriented domain [6, 2].

Data integration solutions on the service-oriented domain deal with query rewriting problems. [5] identifies trends and open issues regarding the use of SLA in data integration solutions on multi-cloud environments. [3] proposes a query rewriting approach which processes queries on data provider services. The query and data services are modeled as RDF views. A rewriting answer is a service composition in which the set of data service graphs fully satisfy the query graph. [4] introduces a service composition framework to answer preference queries. In that approach, two algorithms based on [3] are presented to rank the best rewritings based on previously computed scores. [2] presents an algorithm based on *MiniCon* that produces and order rewritings according to user preferences. The user preferences on this approach are scores used to rank services that should be previously define by the user. Our approach differs from these works in three aspects: (i) the user can express quality measures and associate them to his queries, such as: *I want to use services with response time less than 2 seconds, price per request less than 1 dollar and location close to my city*; (ii) the user preferences guides the service selection. These preferences are matched with the services' quality aspects that are extracted from service level agreement contracts. Here, it is important to highlight that there is a previous phase in which the services' quality aspects are processed and extracted from SLAs. In our proposal we are assuming that these information are accessible and well-formatted to the algorithm; and (iii) the user preferences are also used to guide the rewriting process. The rewriting answers (services compositions) produced must be in accordance with the user preferences. Summarizing, the main and original proposal of our work is to use SLA to guide the entire data integration process.

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.

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

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** $SelectCandidateServices(Q, \mathcal{S})$
2: $\mathcal{L}_{\mathcal{S}} \leftarrow \emptyset$
3: **for all** S_i in $\mathcal{S}$ **do**
4:    **if** $SatisfyQualityMeasures(Q, S_i)$ **then**
5:       $b \leftarrow true$
6:       **for all** A_j in S_i **do**
7:          **if** $Q.notContains(A_i)$ **then**
8:             $b \leftarrow false$
9:             $break$
10:         **end if**
11:       **end for**
12:       **if** $b = true$ **then**
13:          $\mathcal{L}_{\mathcal{S}} \leftarrow \mathcal{L}_{\mathcal{S}} \cup \{S_i\}$
14:       **end if**
15:    **end if**
16: **end for**
17: **return** $\mathcal{L}_{\mathcal{S}}$
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 os 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** $CreateCSDs(Q, \mathcal{L}_\mathcal{S})$
2:   $\mathcal{L}_{\mathcal{CSD}} \leftarrow \emptyset$
3: **for all** S_i in $\mathcal{L}_\mathcal{S}$ **do**
4:   **if** There are mappings h and φ from S_i to Q **then**
5:     $G \leftarrow \emptyset$
6:     $P \leftarrow \emptyset$
7:     **for all** A_j in S_i **do**
8:       $G \leftarrow G \cup \{A_j\}$
9:     **end for**
10:     **for all** M_k in S_i **do**
11:       $P \leftarrow P \cup \{M_k\}$
12:     **end for**
13:     $CSD := \langle S_i, h, \varphi, G, P \rangle$
14:     $\mathcal{L}_{\mathcal{CSD}} \leftarrow \mathcal{L}_{\mathcal{CSD}} \cup \{CSD\}$
15:   **end if**
16: **end for**
17: **return** $\mathcal{L}_{\mathcal{CSD}}$
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 $\mathcal{T}_{\mathrm{init}} [\![\, \mathcal{A}gg(Q)\,]\!]$, $\mathcal{T}_{\mathrm{cond}} [\![\, \mathcal{A}gg(Q)\,]\!]$ and $\mathcal{T}_{\mathrm{inc}} [\![\, \mathcal{A}gg(Q)\,]\!]$. 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.

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

1: **function** $isRewriting(Q, p)$
2: $\quad$ **let** $p = \{CSD_1, CSD_2, ..., CSD_k\}$
3: $\quad$ **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: $\quad\quad$ **return** $true$
5: $\quad$ **end if**
6: $\quad$ **return** $false$
7: **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 analyse the algorithm's behavior. We will present two experiments: *experiment 1* (figure 1) and *experiment 2* (figure 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.
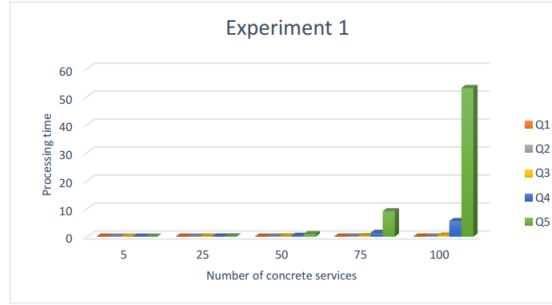

Fig. 1: Query rewriting evaluation.

In the *experiment 1*, 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 (REF WORK): increasing the processing time when the size of the query and the number of concrete services increase.

The *experiment 2* presents the results considering our contribution regarding the use of user preferences and services' quality aspects extracted from SLAs to guide the service selection and query rewriting. *Test 1* and *Test 2* include queries with six abstract services. The important difference between them is use of quality measures guiding the process. *Test 1* do not consider quality measures as any other existing rewriting approach. On the other hand, *Test 2* considers them. The figure 2 shows our results.
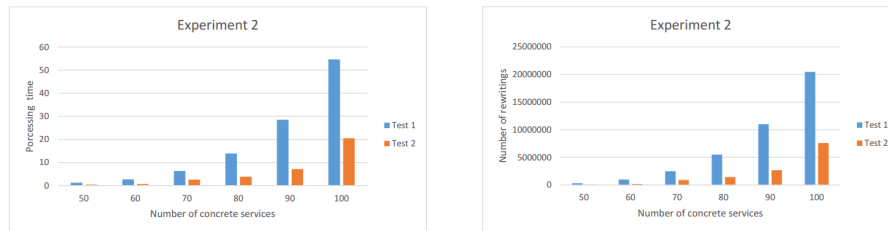

Fig. 2: Here are two figures side-by-side.

Analyzing the *experiment 2* (figures 2 and 3), the results while considering the quality measures are promising. 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.

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. Alhamad, M., Dillon, T.S., Chang, E.: A survey on sla and performance measurement in cloud computing. In: OTM Conferences (2). Lecture Notes in Computer Science, vol. 7045, pp. 469–477. Springer (2011), http://dblp.uni-trier.de/db/conf/otm/otm2011-2.html#AlhamadDC11
2. 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)
3. 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)
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. Da Costa, U., Halfeld Ferrari, M., Musicante, M., Robert, S.: Automatic Refinement of Service Compositions. In: Daniel, F., Dolog, P., Li, Q. (eds.) ICWE. vol. 7977, pp. 400–407. Springer, Aalborg, Denmark (Jul 2013), https://hal.inria.fr/hal-00861101
7. 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

8. 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
9. Leitner, P., Michlmayr, A., Rosenberg, F., Dustdar, S.: Monitoring, prediction and prevention of sla violations in composite services. In: Web Services (ICWS), 2010 IEEE International Conference on. pp. 369–376 (July 2010)
10. 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
11. Maarouf, A., Hamlaoui, M.E., Marzouk, A., Haqiq, A.: Combining multi-agent systems and mde approach for monitoring sla violations in the cloud computing. In: Cloud Technologies and Applications (CloudTech), 2015 International Conference on. pp. 1–6 (2015)
12. Mavrogeorgi, N., Alexandrou, V., Gogouvitis, S., Voulodimos, A., Kiriazis, D., Varvarigou, T., Kolodner, E.K.: Customized slas in cloud environments. In: P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013 Eighth International Conference on. pp. 262–269 (Oct 2013)
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. Rak, M., Suri, N., Luna, J., Petcu, D., Casola, V., Villano, U.: Security as a service using an sla-based approach via specs. In: Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on. vol. 2, pp. 1–6 (2013)