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 de Lyon, CNRS, IAE - Université Lyon 3, LIRIS, UMR5205, 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 offered by the cloud architecture. The quality conditions and

penalties under which these services are delivered can be 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, cloud services (from one or several cloud providers) are chosen to retrieve, process and integrate 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 his/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 the 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. Inspired by these problems and Carrying on the ideas presented in a previous work [3], the aim of this paper is to introduce our service-based query rewriting algorithm guided by user preferences and SLAs which enhances the quality on the results integration in a multi-cloud context.

This paper is organized as follows. Section 2 discusses related works. Section 3 describes the *Rhone* algorithm and its formalization. Experiments and results are described in the section 4. Finally, section 5 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]. Researches have proposed their works addressing this issue [6, 8]. Moreover, once query rewriting is strictly related to data integration, rewriting algorithms have been presented [1, 2, 4].

In [6], the authors introduced a system (called SODIM) which combines data integration, service-oriented architecture and distributed processing. The novelty of their approach is that they perform data integration in service oriented contexts, particularly considering data services. 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. [8] proposed 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. Thus, the query plan executor decides the best location to execute the sub-query to meet privacy and cost constraints. This work is mostly interested in privacy and performance issues forgetting other users' integration requirements.

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 [7]. 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 are presented to rank the best rewritings based on previously computed scores. [1] 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. Furthermore, they do not take into consideration user's integration requirements which can lead to produce rewritings that are not satisfactory in terms of quality requirements and constraints imposed by the user and the cloud environment. We assume that these requirements and constraints be expressed on SLAs. In the next section, we introduce our query rewriting algorithm that deals with SLAs while selecting, filtering and producing results.

3 Rhone service-based query rewriting algorithm

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

3.1 Preliminaries

The idea behind our algorithm consists in deriving a set of service compositions that fulfill the users' integration requirements and constraints 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 requirements and constraints*.

Definition 1 (*Abstract service*). An *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*; and $\overline{I}$ and $\overline{O}$ are a set of comma-separated input and output parameters, respectively. The table 1 exemplifies *abstract services* in a medical scenario. The decorations ? and ! differentiate input and output parameters, 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*.

Definition 2 (*Concrete service*). A *concrete service* is defined as a set of *abstract services*, and by its *quality features* 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. 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 features $(M_1, M_2, .., M_g)$ that 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* $A1$. a and b are *head* variables. *Availability* and *price per call* are identifiers associated to $S1$'s quality features 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 requirements* 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 requirements* 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 requirements can be of two types, single and composed. Single are associated directly to each service involved in the composition. Composed are linked to the entire composition. They are defined in terms of single requirements. 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 a medical scenario 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.

The input data for the *Rhone* is a query 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: selecting candidate concrete services, creating candidate service descriptions and combining and producing rewritings. In the next sections, each step of the algorithm will be described.

3.2 Selecting candidate concrete services

While selecting services, the algorithm deals with three matching problems: (*i*) *quality features* matching, each *feature* in a query should be found in a concrete service. Moreover, the evaluation of a *feature* in a concrete service must satisfy the evaluation of a *feature* in the query; (*ii*) *abstract service* matching, abstract services can be matched if they have the same abstract function name and if the number and type of variable are equivalent; and (*iii*) *concrete service* matching, all abstract services in the concrete service must exist in the query, and

all of them should satisfy the *feature* and *abstract service matching* problems. Compared to [1], our algorithm includes the *features* matching and extends the *concrete service* matching by not accepting *concrete services* that covers useless *abstract services* to the query rewriting.

3.3 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.
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 of *quality features* 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 an output local variable is used as input in another abstract service.

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.

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 follows: $\langle S2,\ h = \{a \rightarrow a,\ b \rightarrow b\},\ \varphi = \{a \rightarrow dis,\ b \rightarrow p\},\ G = \{A1\},\ P = \{availability > 98\%,\ price\ per\ call = 0.1\$\}\rangle$. 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.

3.4 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 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. The *Rhone* algorithm verifies if a given CSD list p is a rewriting of the original query. The function return *true* if (i) the number of abstract services resulting from the union of all CSDs in p is equal 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.

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.

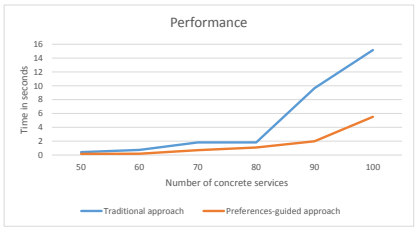
- p_1 and p_2 are not valid rewritings; their number of abstract services do not match with the number of abstract services in the query.
- p_3 is a valid rewriting; the number of abstract services matches and there is no repeated abstract service.

4 Evaluation

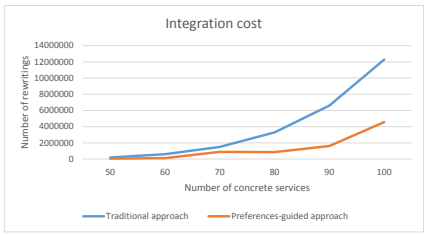
Different experiments were produced to analyze the algorithm's behavior. The *Rhone* [6], so far, was evaluated in a local controlled environment simulating a mono-cloud including a service registry of 100 concrete services. Some experiments were produced to analyze the its behavior concerning performance, 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 1a and 1b summarize our first results.

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 1a). 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 1b).

[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.



(a) Performance evaluation.



(b) Integration cost.

Fig. 1: *Rhone* execution evaluation.

5 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* 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. ElSheikh, G., ElNainay, M.Y., ElShehaby, S., Abougabal, M.S.: SODIM: Service Oriented Data Integration based on MapReduce. Alexandria Engineering Journal (2013)
7. Halevy, A.Y.: Answering queries using views: A survey. The VLDB Journal 10(4), 270–294 (Dec 2001)
8. 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)