Query Rewriting Algorithm for Data Integration Quality

Daniel A. S. Carvalho
Univ. Jean Moulin Lyon 3
Lyon, France
danielboni@gmail.com

Plácido A. Souza Neto
Instituto Federal do Rio Grande do Norte - IFRN
Natal, Brazil
placido.neto@ifrn.edu.br

Chirine G. Guegan
Univ. Jean Moulin Lyon 3
Lyon, France
chirine.ghedira-guegan@univ-lyon3.fr

Nadia Benani
CNRS-INSA
Lyon, France
nadia.bennani@insa-lyon.fr

Genoveva Vargas-Solar
CNRS-LIG-LAFMIA
Grenoble, France
genoveva.vargas@imag.fr

ABSTRACT

This paper introduces the general lines of a rewritring algorithm named *Rhone* that addresses query rewriting for data integration in a multi-cloud environment. The originality of *Rhone* is that the rewriting process is guided by quality measures associated to data providers (services) and user preferences including their subscriptions to the clouds. The paper uses a running scenario to describe the *Rhone*'s implementation and gives some hints about its experimental evaluation.

Keywords

Query rewriting, Data integration, Services composition

1. INTRODUCTION

Data integration has evolved with the emergence of data services that deliver data under different quality conditions related to data freshness, cost, reliability, availability, among others. Data are produced continuously and on demand in huge quantities and sometimes with few associated meta-data, which makes the integration process more challenging. Some approaches express data integration as a service composition problem where given a query the objective is to lookup and compose data services that can contribute to produce a result. Finding the best service composition that can answer a query can be computationally costly. Furthermore, executing the composition can lead to retrieve and process data collections that can require important memory, storage and computing resources. This problem has been addressed in the service-oriented domain [2, 3, 1]. Generally, these solutions deal with query rewriting problems. [2] proposed a query rewriting approach which processes queries on data provider services. [3] introduced a service composition framework to answer preference queries. In that approach, two algorithms based on [2] are presented to rank the best rewritings based on previously computed scores.

[1] presented an algorithm that produces and order rewritings according to user preferences. Yet, to our knowledge few works consider quality measures associated both to data services and to user preferences in order to guide the rewriting process. In the context of the cloud, and also considering that queries can be executed in conditions in which energy, data transfer economic cost, memory and computing resources consumption can be limited or controlled by some economic model, it is important to design approaches and rewriting algorithms that consider these constraints.

This paper introduces the early stages of our ongoing work on developing the *Rhone* service-based query rewriting algorithm guided by SLA's. Our work addresses this issue and proposes the algorithm *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.

The remainder of this paper is organized as follows. Section 2 describes the algorithm *Rhone*, proposed in our work. Section 3 describes a running scenario and also implementation issues. Finally, section 4 concludes the paper and discusses our work perspectives.

2. SERVICE-BASED QUERY REWRITING ALGORITHM

This section describes *Rhone* the service-based query rewriting algorithm that we propose. Given a set of *abstract services*, a set of *concrete services*, a *user query* and a set of user *quality preferences*, derive a set of service compositions that answer the query and that fulfill the quality preferences.

The input for Rhone is: (1) a query; (2) a list of concrete services defined in the following lines.

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:

$$Q(\overline{I}, \overline{O}) := \\ A_1(\overline{I}, \overline{O}), A_2(\overline{I}, \overline{O}), .., A_n(\overline{I}, \overline{O}), C_1, C_2, .., C_m[P_1, P_2, .., P_k]$$

The left side of the definition is called the *head* of the query; and the right side is called the *body.* $\overline{I}$ and $\overline{O}$ are a set of *input* and *output* parameters, respectively. Input parameters in both sides of the definition are called *head variables.* In contrast, input parameters only in the query body are called *local variables.*

Definition 2 (Concrete service) (S):

$$S(\overline{I},\overline{O}) := A_1(\overline{I},\overline{O}), A_2(\overline{I},\overline{O}), .., A_n(\overline{I},\overline{O})[P_1, P_2, .., P_k]$$

A concrete service (S) is defined as a set of abstract services (A), and by its quality constraints P. These quality constraints associated to the service represent the Service Level Agreement (SLA) exported by the concrete service.

The algorithm consists in four steps: (i) select candidate concrete services; (ii) create mappings from concrete services to the query (called *concrete service description (CSD)*); (iii) combine the list of CSDs; and finally (iv) produce rewritings from the query Q.

Algorithm 1 - RHONE

```
 1: function $rhone(Q, \mathcal{S})$
 2: $\mathcal{L_S} \leftarrow SelectCandidateServices(Q, \mathcal{S})$
 3: $\mathcal{L_{CSD}} \leftarrow CreateCSDs(Q, \mathcal{L_S})$
 4: $I \leftarrow CombineCSDs(Q, \mathcal{L_{CSD}})$
 5: $R \leftarrow \emptyset$
 6: $p \leftarrow I.next()$
 7: while $p \neq \emptyset$ and $\mathcal{T}_{\text{init}}[\![ \mathcal{A}gg(Q) ]\!]$ do
 8:     if $isRewriting(Q, p)$ then
 9:         $R \leftarrow R \cup Rewriting(p)$
10:         $\mathcal{T}_{\text{inc}}[\![ \mathcal{A}gg(Q) ]\!]$
11:     end if
12:     $p \leftarrow I.Next()$
13: end while
14: return $R$
15: end function
```

Select candidate concrete services: This step consists in looking for concrete services that can be matched with the query (line 2). In this sense, there are three matching problems: (i) *abstract service matching*, an abstract service A can be matched with an abstract service B only if (a) they have the same name, and (b) they have a compatible number of variables; (ii) *measure matching*, all *single measures* in the query must exist in the concrete service, and all of them can not violate the measures in the query ; and (iii) *concrete service matching*, a concrete service can be matched with the query if all its abstract services satisfy the *abstract service matching* problem and all the *single measures* satisfy the *measures matching* problem.

The result of this step is a list of *candidate concrete services* which may be used in the rewriting process.

Creating concrete service descriptions: In this step the algorithm tries to create *concrete services description* (CSD) to be used in the rewriting process (line 3). A CSD maps abstract services and variables of a concrete service to abstract services and variables of the query. A CSD is created according to variable mapping rules mainly based on 2 criterias: the type and the dependency (variables used as inputs on other abstract services). *Head* and *local* variables in concrete services can be mapped to *head* or *local variables* in the query if they are of the same type. *Local* variables in concrete services can be mapped to *local* variables in the query if: (a) they are of the same type; and (b) the concrete service covers all abstract services in the query that depend on this variable. The relation "depends" means that this *local* variable is used as input in another abstract service. The result of this step is a list of CSDs.

Combining CSDs. Given all produced CSDs (line 4), they are combined among each other to generate a list of lists of CSDs, each element representing a possible composition.

Producing rewritings. In the final step, given the list of lists of CSDs, the algorithm identifies which lists of CSDs are a valid rewritings of the user query (lines 5-13). A combination of CSDs is a valid rewriting iff: (i) the number of *abstract services* in the query is equal to the result of adding the number of *abstract services* of each CSD; (ii) there is no duplicated abstract service; (iii) there is mapping to all head variables in the query; and (iv) if the query contains a *composed measure*, that corresponds to the preferences associated to the query. Every element in the CSD list has its corresponding *composed measure* (represented as the called function *isRewriting*(Q, p) - line 8). The result of this step is the list of valid rewritings of the query (line 14), that is those the provide expected data and respect quality preferences.

3. IMPLEMENTATION AND RESULTS

Let us suppose the following medical scenario to illustrate our service-based query rewriting algorithm. Users can retrieve information about patients, diseases, dna information and others. To perform these function consider the *abstract services* in table 3. *Abstract services* are a set of basic service capabilities.

Abstract Service	***Description***
DiseasePatients(d?,p!)	Given a disease *d*, a list of patients *p* infected by it is retrieved.
PatientDNA(p?,dna!)	Given a patient *p*, his DNA information *dna* is retrieved.
PatientInformation(p?,info!)	Given a patient *p*, his personal information *info* is retrieved.

Table 1: List of *abstract services*

In our scenario, a *query* expresses an abstract composition that describes the requirements of a user. *Queries* and *concrete services* are defined in terms of *abstract services.* They can be associated to a single *abstract service* or to a composition of them.

Let us consider the following query: *a user wants to retrieve patient's personal and DNA information of patients who were infected by a disease 'K' using services that have availability higher than 98%, price per call less than 0.2 dollars, and total cost less then 1 dollar.*

A query Q tagged with user preferences is defined in accordance with the grammar:

$$Q(\overline{I},\overline{O}) := A_1(\overline{I},\overline{O}), A_2(\overline{I},\overline{O}), .., A_n(\overline{I},\overline{O})[P_1, P_2, .., P_k]$$

where the left side is the *head* of the query; and the right side is the *body.* $\overline{I}$ and $\overline{O}$ are a set of *input* and *output* parameters, respectively. Input parameters present in both sides of the definition are called *head variables.* In contrast, input parameters only in the body are called *local variables.* $A_1, A_2, .., A_n$ are *abstract services.* $P_1, P_2, .., P_k$

are user preferences (over the services). Preferences are in the form $x \otimes constant$ such that $\otimes \in \{\geq, \leq, =, \neq, <, >\}$. The query which express the example following our grammar is below. The decorations ? and ! are used to specify input and output parameters, respectively.

$$Q(d?, dna!) := DiseasePatients(d?, p!), PatientDNA(p?, dna!), \\ [availability > 99\%,\ price\ per\ call < 0.2\$,\ total\ cost < 1\$]$$

We highlight that in the query there are two types of preferences (let's refer to them as *measures*): *single measures* (availability and price per call) and *composed measures* (total cost). The *single measures* are the simplest type. It is a static measure which is has a name associated with an operation and a value. The *composed measure* is dynamically computed measure. It is defined as aggregations of *single measures*.

Concrete services are defined follwing the same grammar as the *query*. The only difference is that concrete services do not have *composed measures*. We use 7 concrete services to run our approach.

In this example all the queries have 6 *abstract services* and 2 *single measures*. The number of local variables (dependencies) and CSDs is being modified to see how the algorithm works under these conditions.

By now, the analysis identified that the factor that influenciates the Rhone performance is the number of CSDs versus the number of abstract services in the query since they increase the number of possible combinations of CSDs.

4. CONCLUSIONS

This work proposes a query rewriting algorithm for data integration quality named *Rhone*. Given a query and a list of concrete services as input, the algorithm looks for candidate concrete services. These candidate services can be used probably in the rewriting process to match the query. The algorithm is implemented in Java and we are currently performing experiments in order to better evaluate the performance of the *Rhone* considering different sets of services in a multi-cloud environment.

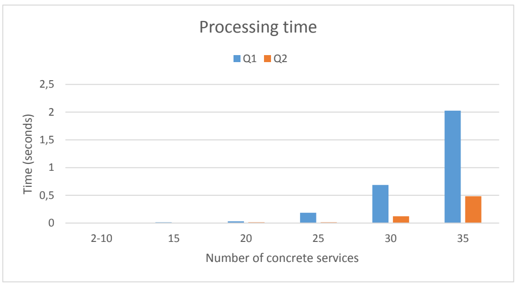
5. REFERENCES

[1] C. Ba, U. Costa, M. Halfeld Ferrari, R. Ferre, M. A. Musicante, V. Peralta, and S. Robert. Preference-Driven Refinement of Service Compositions. In *CLOSER (4th International Conference on Cloud Computing and Services Science)*, Proceedings of CLOSER 2014, page 8 pages, Barcelona, Spain, Apr. 2014. POSTER presentation.

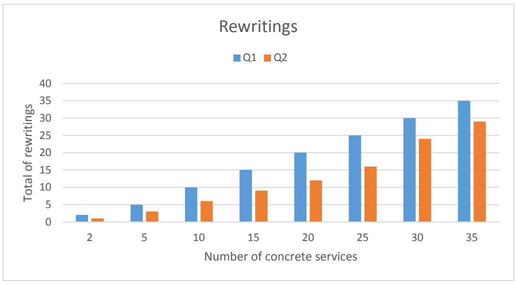
[2] M. Barhamgi, D. Benslimane, and B. Medjahed. A query rewriting approach for web service composition. *Services Computing, IEEE Transactions on*, 3(3):206–222, July 2010.

[3] K. Benouaret, D. Benslimane, A. Hadjali, and M. Barhamgi. FuDoCS: A Web Service Composition System Based on Fuzzy Dominance for Preference Query Answering, Sept. 2011. VLDB - 37th International Conference on Very Large Data Bases - Demo Paper.

[4] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, Dec. 2001.



(a) Droopy



(b) Snoop

Figure 1: Subfiguras