Alternate *ACM* SIG Proceedings Paper in LaTeX Format

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

Placido A. Souza Neto
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
???????
Grenoble, France
genoveva.vargas@imag.fr

ABSTRACT

...

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

ACM proceedings, LaTeX, text tagging

1. INTRODUCTION

Data integration is a well-known and widely discussed problem in the database domain. The emergence of new architectures like the cloud opens new opportunities for data integration. The possibility of having unlimited access to cloud resources and the "pay as U go" model make it possible to change the hypothesis for processing big data collections.

Existing data integration techniques must be revisited considering weakly curated and modeled data sets provided by different services under different quality conditions. Data integration can be done according to (i) quality of service (QoS) requirements expressed by their consumers and (ii) Service Level Agreements (SLA) exported by the cloud providers that host huge data collections and deliver resources for executing the associated management processes. Yet, it is not an easy task to completely enforce SLAs particularly because consumers use several cloud providers to store, integrate and process the data they require under the specific conditions they expect. For example, 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 [**?**]. Naturally, a collaboration between cloud providers becomes necessary [**?**] but this should be done in a user-friendly way, with some degree of transparency.

In this context, the aim of our work is to present the early stages of our ongoing work on developing the Rhone service-based query rewriting algorithm.

The remainder of this paper is organized as follows. Section **??** describes ... . Section **??** ... . Section **??** ... .

2. SCENARIO AND PRELIMINARIES

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 2. *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.

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

It is important to 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*. To illustrate our approach in the next section, consider the *concrete services* in table 2.

$S1(a?, b!) := DiseaseInfectedPatients(a?, b!)$ $[availability > 99\%,\ price\ per\ call = 0.2\$]$
$S2(a?, b!) := DiseaseInfectedPatients(a?, b!)$ $[availability > 99\%,\ price\ per\ call = 0.1\$]$
$S3(a?, b!, c!) := DiseaseInfectedPatients(a?, b!, c!)$ $[availability > 98\%,\ price\ per\ call = 0.1\$]$
$S4(a?, b!) := PatientDNA(a?, b!)$ $[availability > 99.5\%,\ price\ per\ call = 0.1\$]$
$S5(a?, b!) := PatientDNA(a?, b!)$ $[availability > 99.7\%,\ price\ per\ call = 0.1\$]$
$S6(a?, b!) := PatientInformation(a?, b!)$ $[availability > 99.7\%,\ price\ per\ call = 0.1\$]$
$S7(a?, b!) := PatientDNA(a?, c!), PatientInformation(c?, b!)$ $[availability > 99.7\%,\ price\ per\ call = 0.1\$]$

Table 2: Available *concrete services*

3. SERVICE-BASED QUERY REWRITING ALGORITHM

The algorithm described in this section is called *Rhone*. The Rhone service-based query rewriting algorithm addresses the problem of 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 fulfill the quality preferences.

The algorithm includes four steps: (i) select candidate concrete services; (ii) create mappings from concrete services to the query (called *concrete service description (CSD)*); (iii) combining CSDs; and (iv) produce rewritings.

Select candidate concrete services. This step consists of looking for concrete services that can be matched with the query. In this sense, there are three matching problems: (i) *abstract service matching*, an abstract service A can be matched with a abstract service B only if: (a) they have the same name. In this case we are assuming they perform the same function; and (b) the number and type of variables should be compatible. This means that the number of input and output variables of A must be equal or higher than the number of input and output variables of B; (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 can be matched with the abstract service in the query (satisfying the *abstract service matching* problem) and all the single measures in the query can be matched with the concrete service measures (satisfying the *measures matching* problem).

The result if this step is a list of *candidate concrete services* which may be used in the rewriting process. Regarding the example, the services selected satisfying the matching rules are: ?????.

Creating concrete service descriptions. In this step of the algorithm tries to create *concrete services description* (CSD) to be used in the rewriting process. 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 the following variable mapping rules: (i) *head variables* in concrete services can be mapped to *head* or *local variables* in the query if they are from the same type; (ii) *local variables* in concrete services can be mapped to *head variables* in the query if they are from the same type; and (iii) *local variables* in concrete services can be mapped to *local variables* in the query if: (a) they are from the same type; and (b) the concrete service cover all abstract service in the query that depends on this variable. Depends here means that this local variable is used as input in another abstract service. As result a list of CSDs is produced. Regarding the example CSDs are created to ?????.

Combining CSDs. In this step, given all CSDs produced, all combinations of them is generated resulting in a list of lists of CSDs.

Producing rewritings. In the final step, given the list of lists of CSDs, the algorithm identifies which lists of CSDs are a valid rewriting of the user query. A combination of CSDs is a valid rewriting if: (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 abstract service in duplicity; (iii) there is mapping to all head variables in the query; and (iv) if the query contains a *composed measure*, this measure must be updated for each rewriting produced, and it can not be violated.

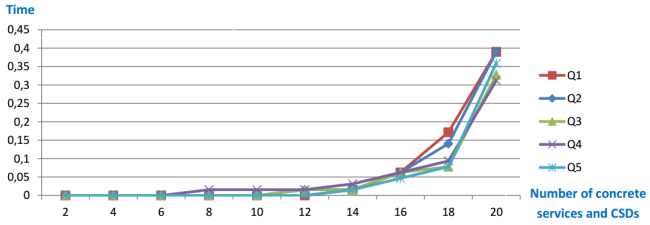
As result of this step we have a list of rewritings of the query. Regarding our example the query has a preference which is associated to the rewritings (composed measure). Its value is updated while rewriting the query. In that case, *total cost* is updated by aggregating the value of *price per call* of each service. The rewritings produced are below. Note that more rewritings can be produced if the *composed measure* did not exists. The rewrintgs are listed in the lexicographical order considering the concrete services.

```
Q(disease?, info!, dna!) := S1(disease?,p!) S7(p?,info!) S4(p?,dna!)
Q(disease?, info!, dna!) := S3(disease?,p!, _) S7(p?,info!) S4(p?,dna!)
Q(disease?, info!, dna!) := S1(disease?,p!) S8(p?,info!) S4(p?,dna!)
```

4. IMPLEMENTATION AND RESULTS

The algorithm is implemented in Java. We are currently performing experiments in order to evaluate the performance of the Rhone. The figure 4 is an example of the charts

we have created. 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 vs. the number of abstract services in the query since they increase the number of possible combinations of CSDs.

5. CONCLUSIONS

...

6. ACKNOWLEDGMENTS

optional..