Rhone: A Query Rewriting Algorithm for Data Integration Quality

Daniel A. Carvalho[1], Plácido A. Souza Neto[a], Chirine Ghedira-Guegan[1], Nadia Bennani[1], Genoveva Vargas-Solar[b]

[a]*Instituto Federal do Rio Grande do Norte – Natal, Brazil*
[b]*CNRS, LIG-LAFMIA, Saint Martin d'Hères, France*

Abstract

Keywords: keyword, keyword.

1. Introduction

2. Related Work

3. Service-based query rewriting algorithm

This section describes the service-based query rewriting algorithm, called *Rhone*, that we propose. 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 that answer the query and that fulfill the quality preferences regarding the context of data service deployment.

The algorithm consist in three macro-steps: (*i*) select services; (*ii*) create variable mappings from services to the query; and (*iii*) produce rewriting that matches with the query. In the following lines, basic definitions are presented to explain and describe each step of the algorithm in detail. The input for the *Rhone* algorithm is: (1) a query; (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:

☆Contact author: Plácido A. Souza Neto
Email address: `placido.neto@ifrn.edu.br` (Plácido A. Souza Neto)
[1]*Address:* IFRN - Avenida Senador Salgado Filho, 1559, Campus Natal Central - Tirol. CEP 59015-000, Natal, RN, Brazil. *Telephone/Fax:* +55 84 4005 9980. *Mobile phone:* +55 84 99911 1931.

$$Q(\overline{I}_h;\overline{O}_h) := A_1(\overline{I}_{1l};\overline{O}_{1l}), A_2(\overline{I}_{2l};\overline{O}_{2l}), .., A_n(\overline{I}_{nl};\overline{O}_{nl}), C_1, C_2, .., C_m[P_1, P_2, .., P_k]$$

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. There are two types of parameters: the ones who appears in the *head* definition called *head* variables, and the ones who appears only in the *body* definition called *local* variables. The sets $\overline{I}_h$ and $\overline{O}_h$ refer to *head* input and output variables, and the sets $\overline{I}_l$ and $\overline{O}_l$ refer to *local* input and output variables. Intuitively, $\overline{I}$ is the union of $\overline{I}_h$ and all $\overline{I}_l$ such as $\overline{I} = \overline{I}_h \cup \{\overline{I}_{1l}, .., \overline{I}_{nl}\}$. The same rule can be applied to output variables: $\overline{O} = \overline{O}_h \cup \{\overline{O}_{1l}, .., \overline{O}_{nl}\}$. *Abstract services* $(A_1, A_2, .., A_n)$ describes a set of basic service capabilities. $C_1, C_2, .., C_m$ are *constraints* over the *input* and/or *output* parameters. 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 $\otimes \in \{\geq, \leq, =, \neq, <, >\}$.

Definition 2 (Concrete service): A concrete service (S) is defined as a set of *abstract services*, and by its *quality constraints* according to the grammar:

$$S(\overline{I}_h;\overline{O}_h) := A_1(\overline{I}_{1l};\overline{O}_{1l}), A_2(\overline{I}_{2l};\overline{O}_{2l}), .., A_n(\overline{I}_{nl};\overline{O}_{nl})[Q_1, Q_2, .., Q_k]$$

A *concrete service* definition is similar to the *query* definition, excepting the fact that a *concrete service* does not have constraints over input and output variables. The input variables $\overline{I}$ is the union of $\overline{I}_h$ (*head* variables) and all $\overline{I}_l$ (*local* variables) such as $\overline{I} = \overline{I}_h \cup \{\overline{I}_{1l}, .., \overline{I}_{nl}\}$, and the output variables $\overline{O} = \overline{O}_h \cup \{\overline{O}_{1l}, .., \overline{O}_{nl}\}$. $Q_1, Q_2, .., Q_k$ are *quality measures* associated to the concrete service. These *measures* reflect the quality aspects present in the service level agreement exported by the concrete service. Q_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 $\otimes \in \{\geq, \leq, =, \neq, <, >\}$.

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 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 measure* P_i in Q, there is one *single measure* Q_i in S that satisfies the *measures* matching problem (Definition 3).

For the *candidate concrete services* selected regarding the matching problems, the algorithm creates mappings from these services to the *query*. A *candidate service description* (CSD) describes how a *candidate concrete service* can be used in the *query* rewriting process.

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 variable mapping rules mainly based on 2 criterias: the type and the dependency (variables used as inputs on other *abstract services*).

Rule 1: *Head* variables in *concrete services* can be mapped to *head* or *local* variables in the *query*.

Rule 2: *Local* variables in *concrete services* can be mapped to *head* variables in the *query*.

Rule 3: *Local* variables in *concrete services* can be mapped to *local* variables in the *query* if 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*.

Lembrar rhone iterator... adicionar sub-algoritmos

Select candidate concrete services: This step looks 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

Algorithm 1 - RHONE

```
 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: $R \leftarrow \emptyset$
 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
```

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: This step 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 onto 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). 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. The final step (lines 5-13) identifies which lists of CSDs are a valid rewritings of the user query given the list of lists of CSDs. A combination of CSDs is a valid rewriting if: (i) they cover all abstract services in the query; and (ii) there is mapping to all head variables in the query (implemented by the function *isRewriting*(Q, p) - line 8). The originality of our algorithm concerns the aggregation function ($\mathcal{A}gg(Q)$). It is responsible to check and increment *composed measures* (if present in the query). This means for each element in the CSD list the value of *composed measure* is incremented (line 10), and rewritings are produced while the values of these measures are respected (line 7). The result of this step is the list of valid rewritings of the query (line 14).

Texto antigo... adicionar funcoes e ligar ao texto existente.

The function $\mathcal{A}gg(\_)$ in Algorithm 1 simply selects the aggregation conditions present in the query. This function is defined as follows:

$$\mathcal{A}gg(\textbf{SELECT } \alpha \textbf{ FROM } \beta \textbf{ WHERE } \gamma \textbf{ SUCH THAT } \delta) = \delta$$

Where $\delta \in \Delta$ is defined by the following grammar rules (where c denotes any constant and x denotes any identifier:

$$\begin{array}{rcl} \Delta & ::= & E = E \mid E \leq E \mid \cdots \mid \Delta \; ; \; \Delta \\ E & ::= & c \mid x \mid \$SUM(x) \mid \$AVG(x) \mid \cdots \mid \\ & & E + E \mid E - E \mid E * E \mid E/E \mid (E) \end{array}$$

The translation functions $\mathcal{T}_{\text{init}}[\![\ \_\ ]\!]$, $\mathcal{T}_{\text{cond}}[\![\ \_\ ]\!]$ and $\mathcal{T}_{\text{inc}}[\![\ \_\ ]\!]$ take the aggregation portion of a query and produce code, respectively, for the initialization, property check and increment of the variables which implement the aggregation part of the query. Notice that these function definitions are overloaded (they take both elements from Δ and E. They are defined as follows (we use the operation "+" for string concatenation and λ is the empty string):

$$\begin{array}{rcll} \mathcal{T}_{\text{init}}[\![\ \delta_1; \ \delta_2\ ]\!] & = & \mathcal{T}_{\text{init}}[\![\ \delta_1\ ]\!] + \text{``;''} + \mathcal{T}_{\text{init}}[\![\ \delta_2\ ]\!] & \\ \mathcal{T}_{\text{init}}[\![\ E_1 \between \ E_2\ ]\!] & = & \mathcal{T}_{\text{init}}[\![\ E_1\ ]\!] + \text{``;''} + \mathcal{T}_{\text{init}}[\![\ E_2\ ]\!] & \textit{where } \between \in \{=, \leq, <, \ldots\} \\ \mathcal{T}_{\text{init}}[\![\ c\ ]\!] & = & \lambda & \\ \mathcal{T}_{\text{init}}[\![\ x\ ]\!] & = & \lambda & \\ \mathcal{T}_{\text{init}}[\![\ \$SUM(x)\ ]\!] & = & \text{``}\textbf{var } x_{sum} = 0\text{''} & \\ \mathcal{T}_{\text{init}}[\![\ \$AVG(x)\ ]\!] & = & \text{``}\textbf{var } x_{sum} = 0 \ ; \ \textbf{var } x_{count} = 0\text{''} & \\ \mathcal{T}_{\text{init}}[\![\ E_1 \odot \ E_2\ ]\!] & = & \mathcal{T}_{\text{init}}[\![\ E_1\ ]\!] + \text{``;''} + \mathcal{T}_{\text{init}}[\![\ E_2\ ]\!] & \textit{where } \odot \in \{+, -, *, /\} \\ \mathcal{T}_{\text{init}}[\![\ (E)\ ]\!] & = & \mathcal{T}_{\text{init}}[\![\ E\ ]\!] & \end{array}$$

The function $\mathcal{T}_{\text{init}}[\![\ \_\ ]\!]$ generates the initialization code for the aggregates in the query. Notice that, for the sake of simplicity, we suppose that there is no repetition among aggregates.

The next definition generates the tests generated for each aggregation condition in the query:

$$
\begin{array}{rcll}
\mathcal{T}_{\text{cond}}[\![\ \delta_1;\ \delta_2\ ]\!] & = & \mathcal{T}_{\text{cond}}[\![\ \delta_1\ ]\!] + \text{“}\wedge\text{”} + \mathcal{T}_{\text{cond}}[\![\ \delta_2\ ]\!] & \\
\mathcal{T}_{\text{cond}}[\![\ E_1 \circledcirc\ E_2\ ]\!] & = & \mathcal{T}_{\text{cond}}[\![\ E_1\ ]\!] + \text{“}\wedge\text{”} + \mathcal{T}_{\text{cond}}[\![\ E_2\ ]\!] & \textit{where } \circledcirc \in \{=, \leq, <, \ldots\} \\
\mathcal{T}_{\text{cond}}[\![\ c\ ]\!] & = & c & \\
\mathcal{T}_{\text{cond}}[\![\ x\ ]\!] & = & x & \\
\mathcal{T}_{\text{cond}}[\![\ \$SUM(x)\ ]\!] & = & \text{“}x_{sum}\text{”} & \\
\mathcal{T}_{\text{cond}}[\![\ \$AVG(x)\ ]\!] & = & \text{“}x_{sum}/x_{count}\text{”} & \\
\mathcal{T}_{\text{cond}}[\![\ E_1 \odot\ E_2\ ]\!] & = & \mathcal{T}_{\text{init}}[\![\ E_1\ ]\!] + \text{“}\odot\text{”} + \mathcal{T}_{\text{init}}[\![\ E_2\ ]\!] & \textit{where } \odot \in \{+, -, *, /\} \\
\mathcal{T}_{\text{cond}}[\![\ (E)\ ]\!] & = & \text{“(”} + \mathcal{T}_{\text{init}}[\![\ E\ ]\!] \text{“)”} &
\end{array}
$$

The function $\mathcal{T}_{\text{inc}}[\![\ \_\ ]\!]$ is responsible for generating the operations that update the variables involved in the aggregation expressions of the query. This function is defined as follows:

$$
\begin{array}{rcll}
\mathcal{T}_{\text{inc}}[\![\ \delta_1;\ \delta_2\ ]\!] & = & \mathcal{T}_{\text{inc}}[\![\ \delta_1\ ]\!] + \text{“;”} + \mathcal{T}_{\text{inc}}[\![\ \delta_2\ ]\!] & \\
\mathcal{T}_{\text{inc}}[\![\ E_1 \circledcirc\ E_2\ ]\!] & = & \mathcal{T}_{\text{inc}}[\![\ E_1\ ]\!] + \text{“;”} + \mathcal{T}_{\text{inc}}[\![\ E_2\ ]\!] & \textit{where } \circledcirc \in \{=, \leq, <, \ldots\} \\
\mathcal{T}_{\text{inc}}[\![\ c\ ]\!] & = & \lambda & \\
\mathcal{T}_{\text{inc}}[\![\ x\ ]\!] & = & \lambda & \\
\mathcal{T}_{\text{inc}}[\![\ \$SUM(x)\ ]\!] & = & \text{“}x_{sum} := x_{sum} + x\text{”} & \\
\mathcal{T}_{\text{inc}}[\![\ \$AVG(x)\ ]\!] & = & \text{“}x_{sum} := x_{sum} + x; x_{count} := x_{count} + 1\text{”} & \\
\mathcal{T}_{\text{inc}}[\![\ E_1 \odot\ E_2\ ]\!] & = & \mathcal{T}_{\text{inc}}[\![\ E_1\ ]\!] + \text{“;”} + \mathcal{T}_{\text{inc}}[\![\ E_2\ ]\!] & \textit{where } \odot \in \{+, -, *, /\} \\
\mathcal{T}_{\text{inc}}[\![\ (E)\ ]\!] & = & \mathcal{T}_{\text{inc}}[\![\ E\ ]\!] &
\end{array}
$$

The function *BuildPCDs*$(Q, \mathcal{S})$ (Algorithm 1, line 2) is responsible for creating one PCD for each concrete service.

The iterator I is responsible for producing sets of PCDs such that *(i)* their combination *covers* the body of the abstract query and *(ii)* they are produced in a decreasing order of priority (the first set produced by the iterator maximizes the combined preference).

The iterator I implements a dynamic priority queue. Its initialization stores the most preferred solutions in the queue (according to the order determined by the SLAs). The operation $I.\textit{Next}()$ de-queues the next preferred set of PCDs, calculates the successors of this set (in the pareto order determined by the SLA), inserts the successors in the priority queue and finally returns the de-queued set.

4. Scenario and Data Integration approach

To illustrate the definition, let us suppose the set of abstract services in Table 4 and the Example 1.

Abstract Service	***Description***
DiseaseInfectedPatients(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.
PatientPersonalInformation(p?,info!)	Given a patient p, his personal information *info* is retrieved.

Table 1: Abstract services description

Example 1: *The user wants to retrieve the DNA information from patients infected by the disease 'K' using services that have availability higher than 99%, price per call less than 0.2 dollars, and the total cost less then 1 dollar.*

The query which express the Example 1 according to the Definition 1 and the abstract services in Table 4 is specified below. The decorations ? and ! are used to specify input and output parameters, respectively.

$$Q(d?, dna!) := DiseaseInfectedPatients(d?, p!), PatientDNA(p?, dna!),\\ d = \text{“}K\text{”}[availability > 99\%,\ price\ per\ call < 0.2\$,\ total\ cost < 1\$]$$

Analyzing the query, it is possible to note that the parameters “d?” and “dna!” appear in both sides of the definition. Due to that they are *head* variables. On the other hand, “p!” and “p?” are *local* variables considering that they appear only in the body definition. Additionally, note that the local variables “p!” and “p?” have the same name. Intuitively, this fact indicates a dependency between the abstract services which use these variables (in that case *DiseaseInfectedPatients* and *PatientDNA*).

In the example, *DiseaseInfectedPatients* and *PatientDNA* are abstract services that specify basic service functions which are combined to answer the query. The constraint ($d = \text{“}K''$) over the input parameter ‘d’ will be further used while executing the query over a database (the where clause). *Availability*, *price per call* and *total cost* are the user preferences over the services.

Example 2: Considering the query (see Example 1) and the abstract services (see Table 4), the concrete services below are examples in accordance with the Definition 2.

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

For example, looking to the concrete services in the Example 2, the abstract service *DiseaseInfectedPatients* in $S1$ and $S2$ are equivalent to the abstract service *DiseaseInfectedPatients* in the query Q (Example 1) once they have the same name and number of input/output parameters. On the other hand, the abstract service *DiseaseInfectedPatients* in $S3$ is not equivalent to the abstract service *DiseaseInfectedPatients* in the query because the number of parameters are different.

For example, considering the query in the Example 1 and the concrete services in the Example 2, it is possible to see that: (1) $S1$ is not a candidate service because it violates an user preference (*price per call*); (2) $S3$ and $S7$ are not a candidate service because they have abstract services that are not in Q; and (3) $S2$, $S4$ and $S5$ are candidate services once: all their abstract services have an equivalent in Q and there is no violation in the user preference.

Example 3: To illustrate the rules above consider the following example. *The user wants to retrieve the personal information and the DNA information from patients infected by disease "K"*. Supposing we have the query Q and the concrete services $S1$, $S2$, $S3$ and $S4$:

$$Q(d?, info!, dna!) :=$$
$$DiseaseInfectedPatients(d?, p!), PatientDNA(p?, dna!), PatientPersonalInformation(p?, info!)$$
$$S1(a?, b!) := DiseaseInfectedPatients(a?, c!), PatientDNA(c?, b!)$$
$$S2(a?, b!) := PatientPersonalInformation(a?, b!)$$
$$S3(a?, b!) := DiseaseInfectedPatients(a?, b!)$$
$$S4(a?, b!) := PatientDNA(a?, b!)$$

In the query Q it is possible to note that "p!" is a *local* variable which is used as input ("p?") for the abstract services $A2$ and $A3$. Looking to the concrete service

$S1$ no CSD will be created for it because the *local* variable $c!$ is mapped to the local variable $p!$, but $S1$ does not cover all abstract services which expects that variable. On the other hand, CSDs are constructed to the services $S2$, $S3$ and $S4$ once even existing the mapping from a local variable in the concrete service to a local variable in the query, all of them only cover one abstract service which uses that *local* variable. To be more clear about these rules, consider the rewriting below in which the CSDs for the services $S2$, $S3$ and $S4$ are used:

$$Q(d?, info!, dna!) := S3(d?, p!), S4(p?, dna!), S2(p?, info!)$$

The rewriting above is the only one possible for the query. However, let us suppose that a CSD for $S1$ was created violating the rule number two, consequently the wrong rewriting below would be created:

$$Q(d?, info!, dna!) := S1(d?, info!), S4(p?, dna!)$$

The problem here is regarding the *local* variable $p?$ which appears in $S4$, and it apparently should come from $S1$, but we can not guarantee that the same *local* variable internally used in $S1$ is the one expected by $S4$. That is the reason the rule two exists.

5. Experiments or Evaluation

6. Conclusions