Optimizing DaaS Web Service based Data Mashups

Mahmoud Barhamgi, Chirine Ghedira, Djamal Benslimane, Salah-Eddine Tbahriti, Michael Mrissa

Claude Bernard Lyon1 University, 69622 Villeurbanne, France
firstname.lastname@liris.cnrs.fr

***Abstract*—Data Mashup is a special class of mashup application that combines information on the fly from multiple data sources to respond to transient business needs. In this paper, we propose two optimization algorithms to optimize Data Mashups. The first allows for selecting the minimum number of services required in the data mashup. The second exploits the services' constraints on inputs and outputs to filter out superfluous calls to component services in the data mashup. These two algorithms are evaluated and tested in the healthcare application domain, and the reported results are very promising.**

***Keywords*—Services composition, Optimization, Query rewriting, Data Services.**

1. Introduction

Data mashup is a special class of mashup application that combines information from several data sources to meet user requests [9,3]. Typically, data sources are provided through Web Services; this type of Web services is known as DaaS (Data-as-a-Service) Web services [5,6]. Data mashup has become so popular over the last few years; its applications vary from addressing transient business needs in modern enterprises [9] to conducting scientific research in e-science communities [3].

Informally speaking, a data mashup represents a "*parameterized query"* over a set of DaaS services. For example, one can "*mash-up*" the DaaS services in Table-1, to answer parameterized queries like: *Q: "what are the medications Y that may interact with a given medication X?",* where the value of *X* is specified by the user at the execution time of the data mashup. To answer *Q*, the services can be composed as follows: one (or more) of the services {S_3, S_4, S_5} need to be invoked with the medication identifier specified by the user, then one (or more) of the services {S_1, S_2} are invoked with the returned interacting medications to retrieve detailed information about those interacting medications.

One of the most challenging problems when answering parameterized queries over DaaSs (i.e. constructing data mashups) is that component services cannot be chosen precisely at the composition time. Consider for example the query *Q*. Suppose also that the mashup creator is interested in studying the medications whose *codes* belong to a specific range of values (both for the specified medications and their interacting ones), say for instance, the medications whose code values are between /100/ and /300/, (i.e. she will invoke the obtained mashup with values inside the range [100, 300]). The plan for the mashup answering the query is shown in Figure-1, and is detailed as follows. The code of the given medication will be used to invoke S_1 and S_2. Though they have similar semantics, both of these services are included in the plan because the value of the medication's code is not known at the mashup creation time; i.e. the plan needs to include both of these services (which cover different input values as specified by the *Constraints* column in Table-1) to cover all potential input values. The services S_3 and S_4 are invoked to retrieve the codes of interacting medications; these two services are both included because the value of the medication's code is not known at the mashup creation time. Note that the service S_5 is excluded because its ranges of accepted input values and returned output values do not intersect with those specified in the query (medications' codes must be inside the range [100, 300]).

Service	Description	Constraints on inputs/outputs
S_1($a,?b,?c)	Returns the name *b* and reference information *c* of a given medication (identified by its code *a*)	$a \leq 200$
S_2($a,?b,?c)		$a \geq 200$
S_3($a,?b)	Returns interacting medications (identified by their codes ***b***) of a given medication (identified by its code ***a***)	$a \leq 200, b \leq 400$
S_4($a,?b)		$a \geq 200, b \leq 400$
S_5($a,?b)		$800 \geq a \geq 600, 800 \geq b \geq 600$

Table-1: A Set of DaaS Web Services

The example shows that multiple similar DaaS services may be mapped to the same part of the query (e.g. S_3, S_4, S_5 have the same semantics, they all return the interacting medications of a given one), but cover different ranges of accepted input values and provide different ranges of output values (see the *Constraints* column in Table-1). The issue the mashup creator is confronted to is how to choose the minimum number of similar services that must be combined together to cover completely (if possible) the value ranges implied by the mashup query. For example, the mashup creator needs to realize that only the services S_3 and S_4 are needed to cover "completely" the corresponding part of the query (see Figure-1); the service S_5 needs to be eliminated because it covers value ranges that are irrelevant to the query. Note that even if it was covering relevant ranges, if S_3 and S_4 were already selected, then S_5 would be *redundant* as the corresponding portion of the query is already completely covered by the combination of S_3 and S_4. By *redundant* we mean that the tuples returned by S_5 and which belong to *Q*, would be also returned by S_3 and S_4. Note that the general case is even harder since DaaS

978-0-7695-4462-5/11 $26.00 © 2011 IEEE
DOI 10.1109/SCC.2011.72

services may have multiple inputs and multiple outputs associated with value constraints that intersect partially with the query's constraints, and the mashup creator should be able to select the "*minimum*" number of services required for his needs. Furthermore, mashup queries may be complex; comprising many sub-queries, matching hundreds of DaaS services, each. This issue is very important since adding services more than needed (e.g. adding S_5 to S_3, S_4) would impact badly the execution time of the mashup without contributing new tuples on the mashup's output. Calls to redundant services are considered as waste of time.

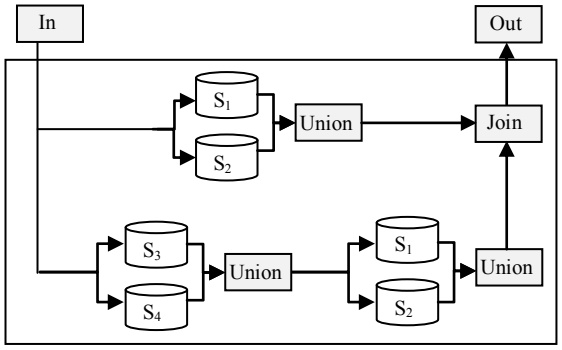

Figure 1: A plan for a data mashup that would return the interacting medications for any given medication with code value within the range [100, 300].

In a previous work [1] we proposed an approach to mash-up DaaS Web services using RDF query rewriting techniques. In this paper, we extend that approach with optimization techniques to speed up the execution of data mashups. The contributions of this paper over our previous work are the following:

- We propose an efficient algorithm to select the minimum number of DaaS services matching the different portions of a mashup query. The algorithm improves considerably the execution time of mashups.
- We improve the efficiency of our previous mashup algorithm by introducing the use of *Virtual RDF Views* and by clustering services *prior to* receiving mashup queries.
- We exploit the services' data value constraints on inputs and outputs to optimize the mashup composition plan.
- We evaluated the performance gain introduced by our algorithms by a set of experiments conducted in the healthcare application domain.

The rest of the paper is organized as follows. In section 2, we extend our approach to data mashup with a virtual view layer to improve its efficiency. We present also our algorithm for the selection of the minimum set of services. In Section 3, we exploit the services constraints to optimize mashups. In Section 4 we evaluate our algorithms. In Sections 5 we review related works. In Section 6 we conclude the paper with our perspectives.

2. A Declarative Approach to Mash-up DaaS Services

In a previous work [1] we presented a generic approach to compose DaaS Web services based on RDF query rewriting techniques. In that work, DaaS services are modeled as RDF views over domain ontologies to capture their semantics. These RDF views are used to annotate the service description files. The approach exploits these views to rewrite the mashup query directly in terms of available DaaS Web services. However, this may not be always desirable because the general problem of query rewriting has a high complexity of the order NP-Complete [8] since it may involve searching through an exponential number of rewritings. This may present an important scalability problem for the mashup algorithm when the number of available DaaS Web services is large. In this paper, we extend our previous approach with a *virtual layer* to increase its scalability. Figure-2 presents the extended data mashup approach. The approach follows a three-level query model detailed as follows:

- ***The Query Level***: This level consists of a set of domain ontologies that give users an interface for formulating and submitting their declarative mashup queries. Mashup queries are expressed in SPARQL, the do facto query language for the Semantic Web.
- ***The Virtual Level***: This level consists of the different "meaningful" parameterized RDF views that can be implemented by concrete DaaS Web services within an application domain. The *virtual RDF views* represent meaningful queries the data holders would share with each other in a given application domain. In the healthcare application domain, the examples include: "*find the interacting medications of a given one*", "*find the different information about a given medication*", etc). The rationale behind the use of the virtual views is that in real-life application domains multiple service providers may provide the same DaaS service (i.e. they implement the same virtual view), perhaps with different quality criteria. Meaningful virtual RDF views may be defined by domain experts and stored by the proposed mashup system.
- ***The Concrete Level***: This level represents the space of the DaaS Web services offered on the Web for a particular domain of application (i.e. the potential candidates for answering data mashup queries). DaaS services in this level are represented by their parameterized RDF views. Concrete DaaS services are clustered in groups; each group represents a virtual parameterized RDF view. A concrete DaaS Web service may further personalize its corresponding virtual RDF view by specifying value constraints (i.e. order and equality constraints, e.g. $a \geq 10$) on its accepted input parameters and the returned output parameters. It may also characterize the quality of the information it provides [17] (via a set of DQ Data Quality metrics).

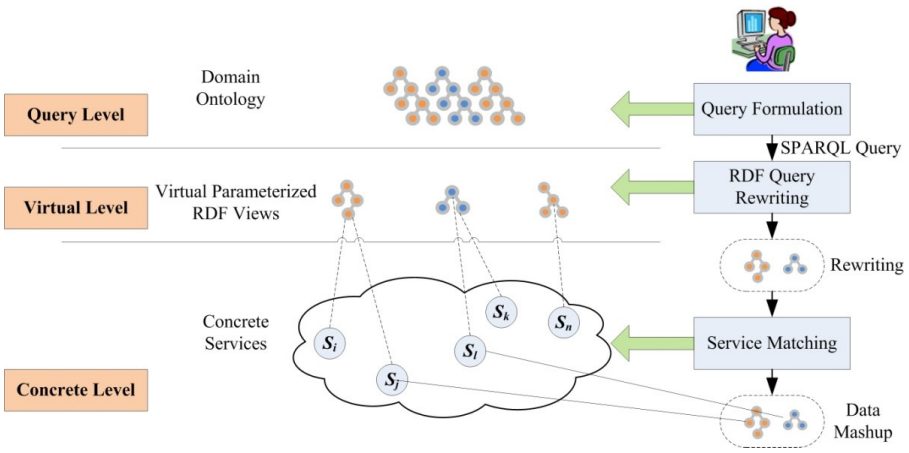

Figure 2: Three-level Query Scheme: (a) Ontologies from the *query level* are used to formulate mashup queries. (b) The query model rewrites queries in term of the *virtual parameterized views* from the *virtual level*. (c) The model then matches virtual views to concrete DaaS Web services on the *concrete level.*

In Figure-2, we illustrate how DaaS services are queried and composed in this data mashup approach. As the figure shows, the user formulates her queries over domain ontologies (using SPARQL query language). Next, the posed mashup query is rewritten in terms of the virtual parameterized RDF views (which represent the meaningful queries in a given application domain) using an RDF query rewriting algorithm that we have devised for that purpose [1]. The virtual RDF views that participate in the obtained rewriting will be matched against the RDF views of concrete services from the concrete space. Matching DaaS services will be used in constructing the mashup that will answer the posed query. In such approach, data mashup queries will be always resolved in a reasonable time due to the following reasons: (*i*) the number of logical views (i.e. the virtual RDF views) is limited compared to concrete DaaS services, (*ii*) it suffices to pick up one rewriting only out of the possible ones and match it to concrete DaaS services− (i.e. the rewriting algorithm can stop as soon as the first rewriting is available), (*iii*) concrete DaaS Web services implementing a given virtual RDF view can be ranked and classified *prior to* query resolution based on their qualities.

We illustrate these steps based on the running example in the following sub sections. Since the paper focuses on the optimization issue by selecting the minimum number of similar services participating in a mashup (during the matching phase), we will show briefly how the mashup query is rewritten in terms of virtual parameterized RDF views (for deeper details about the rewriting algorithm the reader is referred to [1]), then we will devote the most of our talk to the matching phase.

2.1. RDF Query Rewriting

In this phase the query is rewritten in terms of the virtual parameterized RDF views. Figure-3 shows the query in the running example along with a set of virtual parameterized RDF views. The view V_1 returns the different information about a given medication, V_2 returns the interacting medications of a given one (input parameters are prefixed with "$", and output parameters with "?"). The query rewriting algorithm in [1] compares the graph of the query to those of the virtual RDF views to determine the portions of the query (i.e. the sub-graphs) that can be covered by individual RDF views. It establishes also the partial containment mappings between the query and the views. Table-2 shows these mappings between the query and the views in the running example. The view V_1 can be used to cover the class-nodes[1] M_{1Q} and M_{2Q} (we use $Q.C_i$ to denote the class-node C_i in the RDF graph of Q); the view V_2 can be used to cover the object property *interacts*. Both combined can be used to cover the query (cover the whole list of class-nodes and object properties of the query). The obtained rewriting is the following:

$Q\ (\$w_1,?y_1,?z_1,?w_2,?y_2,?z_2)\ :-\ V_1(\$w_1, ?y_1,?z_1) \wedge V_2(\$w_1,?w_2) \wedge V_1(\$w_2, ?y_2,?z_2)$

Virtual Views	Variables Mapping	Covered nodes and Object properties
$V_1(\$w_1, ?y_1,?z_1)$	$M_{1Q}\rightarrow M_{V1}$, $w_1\rightarrow a$, $y_1\rightarrow b$, $z_1\rightarrow c$	$M_{1Q}(\$w_1, ?y_1,?z_1)$
$V_1(\$w_2, ?y_2,?z_2)$	$M_{2Q}\rightarrow M_{V1}$, $w_2\rightarrow a$, $y_2\rightarrow b$, $z_2\rightarrow c$	$M_{2Q}(\$w_2, ?y_2,?z_2)$
$V_2(\$w_1,?w_2)$	$M_{1Q}\rightarrow M_{1V2}$, $M_{2Q}\rightarrow M_{2V2}$, $w_1\rightarrow a$, $w_2\rightarrow b$	$Interacts(M_{1Q}, M_{2Q})$

Table-2: The query sub graphs that are covered by virtual RDF views along with the variables mappings

2.2. Service Matching

In this phase the virtual parameterized RDF views participating in the obtained rewriting are matched against the RDF views of concrete services.

[1] A "*Class-node*" is a variable in a query whose type is a class in the domain ontology.

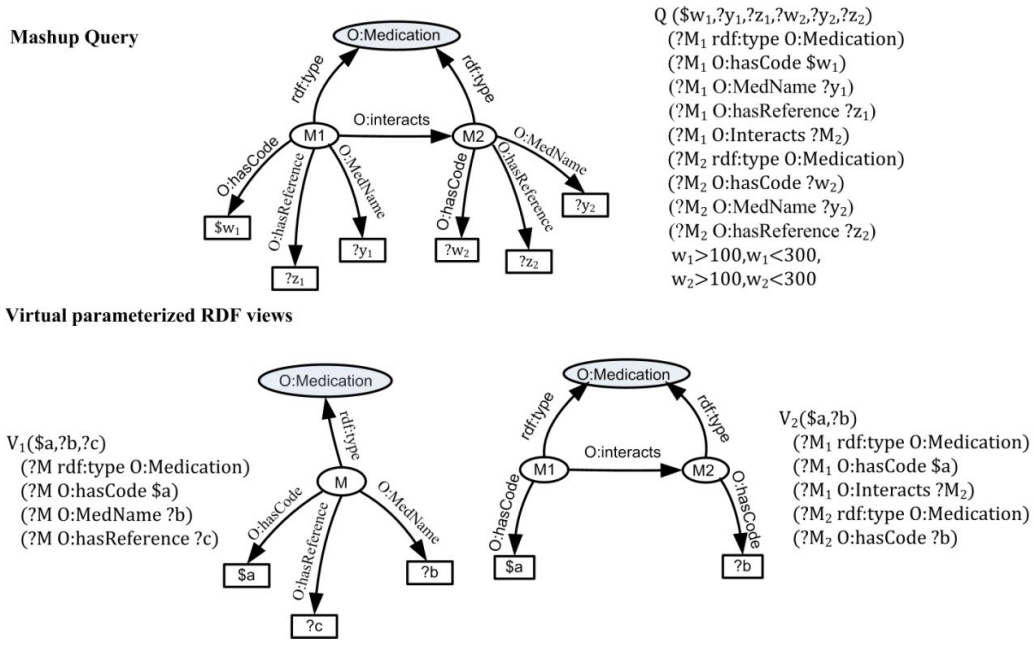

Figure-3: A graphical representation of the data mashup query and the virtual RDF views. Blue ovals are *Concepts* in domain ontologies; White ovals are *Class-nodes*; Arcs are *Datatype/Object properties* in domain ontologies.

The purpose is to select candidate services that can be used to build the data mashup.

Note that if services were not categorized according to the implemented virtual RDF views, the query containment test (i.e. the query subsumption test) in relation with domain ontology [4] would be used to determine whether a service and virtual view are matching each other. In our case, services are clustered according to the implemented views "*prior to*" receiving data mashup queries; i.e. the query containment test is applied by the time services are published to the mashup system. Therefore it suffices in this step to select the minimum number of similar services that must be combined together to cover the data value constraints for each of the individual virtual views used in the rewriting (recall that multiple matching services may exist for the same virtual RDF view, covering different data values for input and output parameter values, each). For example, in the running example, the services S_3, S_4 and S_5 are returned as matches for the view V_2. The issue now is to determine the minimum number of services that entirely cover a virtual view (e.g. V_1, V_2). In case of partial covering (i.e. available services cover only partially a virtual view), the user should be notified so that she knows she should not be waiting for complete answers to her query.

We propose a two-phase algorithm to determine the minimum set of services covering a virtual view efficiently. The algorithm is based on the observation that services can be represented spatially where constrained inputs and outputs are the dimensions of the representation space. This way services can be seen as convex polyhedrons, and the problem of determining if a virtual view is covered by a set of matching services amounts to checking whether a convex polyhedron is subsumed by a finite union of convex polyhedrons. This problem is known to be NP-Complete [11]. We propose a best-effort subsumption algorithm to answer the problem efficiently. Hereafter we detail our two-phase algorithm.

Sketch of our two-phase algorithm:

To illustrate our algorithm, consider the services S_3, S_4 and S_5. They match the virtual view V_2 (in practical cases there may be hundreds of services returned as matches). The spatial representations of these services along with their corresponding virtual view are presented in Figure-4. Input and output parameters are used as the space dimensions. Note that in the general case there may be K constrained input parameters and L constrained output parameters, thus leading to a space with $(K+L)$ dimensions. The figure shows that the services S_3 and S_4 can be jointly used to completely cover the view V_2, and that the service S_5 is irrelevant (there is no intersection between S_5 and V_2).

The proposed algorithm consists of two phases. In the first phase, for each individual virtual view in the rewriting, the algorithm considers a set of N matching services (the value of N can be fixed by mashup creators) and eliminates duplicate (redundant) services and the services that are irrelevant to the virtual view (i.e. those that do not intersect with the view V, e.g. the service S_5). By the end of this phase, the considered set of matching services will contain M services ($M = N$ - the number of eliminated services). In the second phase, the algorithm tests whether the view is completely subsumed by the reduced set of matching services M. In case of partial subsumption, it will add new matching services to M and

will restart from scratch–this is repeated till the considered view is completely covered or there are no more matching services to add for the virtual view at hand. The *M* services in the last iteration constitute the minimum set of services for the considered virtual view.

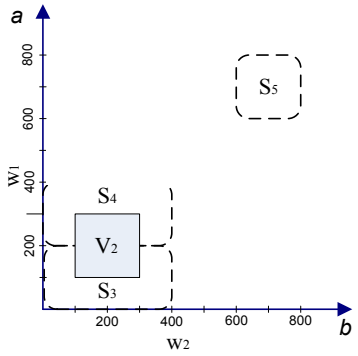

Figure-4: The Graphical Representation of the Constraints Specified on S_3, S_4, S_5 and V_2

The 1st Phase: ***Eliminating Redundant and Irrelevant Services***— The algorithm starts with constructing a table called the *Conflict Table* which points out conflicting and not covered intervals between a view and a set of matching services *G*. Assuming that both *V* and *G* pose constraints in the form of predicates (e.g. *ct*: $x_1 \geq x \geq x_2$) on a number *m* of their inputs/outputs, assume also we have a *k* service, a conflict table is defined as follows.

Definition: *A conflict table T is a k × (2.m) table relating V to all of the predicates defined by the set of services G. An element in T, T_i^j, is $\neg ct_i^j$ if $ct_V^j \wedge \neg ct_i^j$ is satisfiable or is "undefined" otherwise.*

Table-3 is the conflict table for our example (we assume that *N*=3). The first row represents a template for the content of the actual conflict table relating V_2 to S_3, S_4, and S_5. For example, the row corresponding to S_3 is constructed as follows: $ct_V^1 \wedge \neg ct_1^1$ is unsatisfied, thus assigned "*undefined*"; $ct_V^2 \wedge \neg ct_1^2$ is satisfied, thus assigned the value $\neg ct_1^2 = a > 200$; $ct_V^3 \wedge \neg ct_1^3$ and $ct_V^4 \wedge \neg ct_1^4$ are unsatisfied thus assigned "*undefined*".

V_2	***a < low***	***a > high***	***b < low***	***b > high***
S_3	*undefined*	a > 200	*undefined*	*undefined*
S_4	a < 200	*undefined*	*undefined*	*undefined*
S_5	a < 600	*undefined*	b < 600	*undefined*

Table-3: The Conflict Table for the Running Example (N=3)

The algorithm then eliminates duplicate services (i.e. services covering the same values ranges) and the services that do not intersect with the view *V*. This task can be accomplished by computing both the number of *defined elements* and the number of *conflict-free elements* for each row in the conflicting table.

Definition *Two defined entries in the table, $T_{i_1}^{J_{i_1}}$ and $T_{i_2}^{J_{i_2}}$ are said to be conflicting if $i_1 \neq i_2$, and $V \wedge T_{i_1}^{J_{i_1}} \wedge T_{i_2}^{J_{i_2}}$ is not satisfiable. A defined entry $T_{i_1}^{J_{i_1}}$ is said to be conflict-free if it does not conflict with any other defined element $T_{i_2}^{J_{i_2}}$, where $i_1 \neq i_2$.*

Conflict free entries are determined by comparing entries from the conflict table related to the same property (input or output), for different services. If a constraint conflicts with any other constraint defined by another service, the entry is conflicting. It is conflict free otherwise. In our example (see table the Table-3) the defined entries for both S_3 and S_4 are conflicting, while those of S_5 are conflict-free.

Proposition[2]: *If the number of conflict-free elements in the i-th row of T, f_i, is greater than or equal to /1/, or the number of defined elements in row i, $t_i > k$, then S_i is redundant.*

The algorithm counts the number of defined elements for each service S_i in the corresponding row, t_i and computes the number of conflict-free elements, f_i. Then, it removes from the set all services for which t_i is equal to or greater than the current number of services in the set. It also removes services that have at least one conflict free element in the corresponding row of the conflict table. These two steps are repeated until there are no more services that fulfill any of the two conditions. The remaining services form the non-reducible cover set *M* for answering the union covering problem. In our example we have $fc_1, fc_2 = 0, fc_3 = 2, t_1, t_2 = 1 (\leq 3)$, therefore S_5 is removed. In the second iteration, still no service has more defined entries than the total number of services ($t_1 = t_2 = 1 \leq 2$) and there are no conflict free entries, thus the algorithm stops. The minimized cover set is $M = \{S_4, S_5\}$.

The 2nd Phase: Testing the Cover— After eliminating irrelevant services, and since the general subsumption is practically unfeasible [11], a probabilistic ("Monte Carlo type") cover-checking algorithm is applied to guess a point in *V* that is a point witness to non-cover for the set of services *M* (a point located in an uncovered area), if such a point is found, then the subsumption problem is solved with a definite *NO*, i.e. *V*'s constraints are unsatisfied with the set *M*, in such case new matching services will be added and the matching algorithm is repeated until the view is covered or there are no new matching services to add. On the other hand, in case a subsumption relationship exists, the algorithm would try in vain to find such a witness. To prevent this situation, a threshold *d* for the number of guesses is defined, and the algorithm may output a probabilistic *YES*, i.e. $V \subseteq M$ with a predefined probability of error (upper bounded by $(1-\lambda)^d$, where λ is the probability that a randomly generated point *P* inside *V* is a point witness to non-cover)[2]. If *V* was not covered by *M* then the set of tested services are not fulfilling the constraints involved in the query and an

[2] Due to space limitations we do not report the proofs in the paper (they are available upon an email request).

indication will be reported to the user. Continuing with our example, the algorithm will find that V_2 is covered by the services S_3 and S_4 and will return *YES* by the end of the test (i.e., no more matching services are added).

3. Optimizing the data mashup plan using service constraints

Executing the mashup obtained in the previous steps is inefficient. The reason is that some DaaS services are called with values of input parameters violating their specified constraints on accepted input values. Indeed each Web service call usually has some fixed overhead, typically parsing SOAP/XML headers and going through the network stack. Therefore, eliminating superfluous calls (i.e. calls with values violating the service's input constraints) will have a significant impact on the execution time of the whole data mashup.

We exploit the constraints placed on the accepted values of input parameters (and which are specified in the RDF views) to filter out superfluous calls to composed services. For example, filters are inserted before calling S_1 and S_2 to verify whether the value of the medication's code is lower than "200" in the case of S_1, and greater than "200" in the case of S_2. In addition, filters are placed on S_4 and S_5 as can be seen in Figure-5.

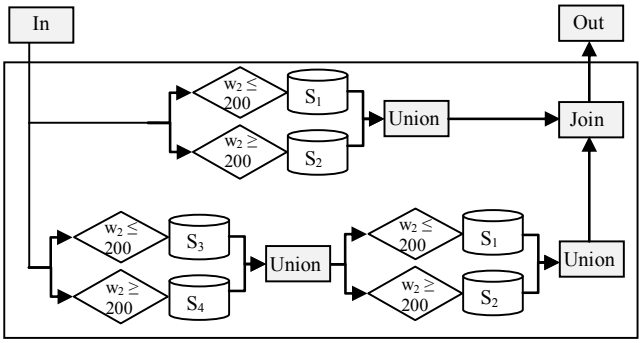

Figure-5: the mashup is optimized using the input constraints as filters before invoking primitive services

In what follows, we show how filters are interpreted in the mashup plan. Similarly to traditional Web services composition, the obtained data mashup is translated into an execution plan describing both data flow and intermediary data processing (e.g. joins, unions, etc) among individual DaaS services. Note that languages like *BPEL4WS* [2] that is used to describe workflow-oriented service compositions cannot be used with data mashups. We translate the mashup plan in terms of set of operations that can be executed by a data streaming execution engine that we have implemented for our purposes. The plan is translated as follows. Each service occurrence in the mashup will be translated to an "*invoke"* operation. Note that an *invoke* operation in our plans is different in nature from those found in BPEL as it invokes the web service for each single tuple in its input relation. The outputs of similar web services (services covering the same portion of the query) will be unified by a "*union"* operation that is responsible for removing redundant tuples. "*Join"* operations will be used to feed a service with data tuples coming from its parents in the mashup plan. "*Select"* operations are used to filter out tuples that do not satisfy a specified constraint (e.g. <, =, > constraints), thus removing superfluous calls to services. It is important to note that data tuples are streamed between the different operations; i.e. our execution engine does not wait until an operation produces all its output tuples to proceed with the execution of subsequent operations, rather an operation starts to execute as soon as its preceding operations begin to produce tuples on their outputs. For space limitation, we don't show the final mashup plan.

4. Evaluation

To evaluate the performance gain obtained from applying our optimization algorithms, we implemented them in our data mashup system and tested the approach in the healthcare domain. In the context of the French project PAIRSE[3], we were provided with access to a set of /411/ DaaS web services implemented on top of /17/ medical databases storing medical information (e.g., medications, diseases, medical tests, allergies, ongoing treatments, etc) about more than /30,000/ patients. The WSDL description files of these services are annotated with RDF views that are defined over a medical ontology to capture their semantics. These services are "*mashed up*" by health actors to answer their daily data needs.

We had two objectives in our conducted experiments: (*i*) we wanted to show that the introduction of virtual views (in the virtual layer) improves the response time of our data mashup system; (*ii*) we wanted to evaluate the cost incurred in finding the minimum number of services matching the virtual views (at the mashup creation time), and see if that cost is justified by the gain obtained in the mashup execution time; we wanted also to evaluate the gain in the mashup execution time introduced by the filter algorithm. All algorithms were implemented in Java, and the reported results in all experiments are the average of /10/ runs.

Objective-I: With help of the healthcare experts we defined the set of virtual RDF views that represent the services. We identified /20/ different virtual views. We clustered our /411/ services according to the virtual views using an OWL-DL-based subsumption test [4] prior to receiving the mashup queries. We considered mashup queries with a varying size (from 3 class-nodes to 10 class-nodes −recall that class-nodes are basic atoms of an RDF query [1]). Figure-6 shows the performance of the mashup query algorithm when (*i*) the queries are rewritten directly in terms of available services (i.e., the initial approach is applied), (*ii*) in terms of the virtual views (i.e.,

[3] https://picoforge.int-evry.fr/cgi-bin/twiki/view/Pairse/Web/

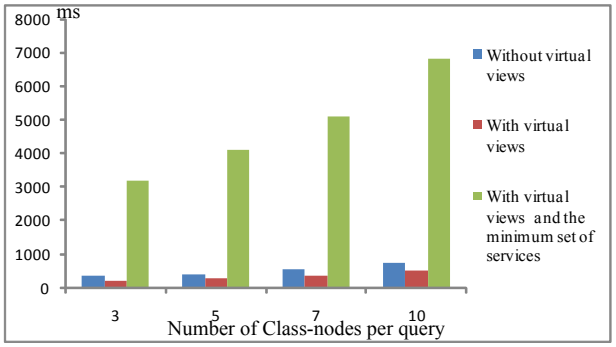

Figure-6: The response time of the mashup system before and after introducing the virtual layer and when the minimum set of services algorithm is applied

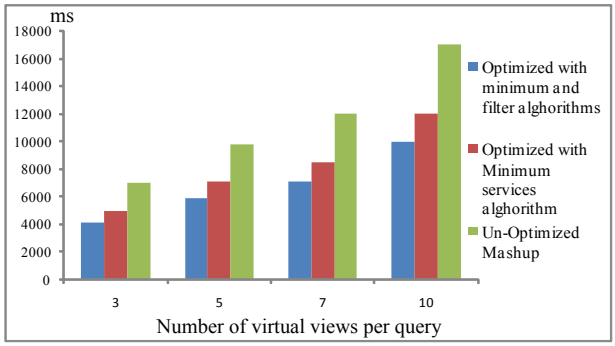

Figure-7: The mashup execution time before and after optimization

the extended approach is applied), and (*iii*) in terms of the virtual views with applying the algorithm of the minimum set of services. Obviously, the use of virtual views improves "considerably" the response time of the mashup system. This gain is due to three factors: (*i*) the number of virtual views is limited compared to concrete services (20 vs. 411), (*ii*) the mashup algorithm is no longer required to go through all potential rewritings; it can stop as soon as a rewriting is found and (*iii*) concrete services are matched (i.e., clustered using the ontology-based subsumption test [4]) to the virtual views "*prior to*" receiving the mashup queries. The results show also that the algorithm of the minimum set of services introduced only a weak overhead at the mashup creation time.

Objective-II: We considered set of mashup queries with varying number of virtual views. We compared the mashup execution time when (*i*) the mashup is not optimized with the minimum set of services algorithm (i.e. the mashup includes some redundant and irrelevant services that would waste time at the mashup execution time), (*ii*) the mashup is optimized with the minimum set of services algorithm and when (*iii*) both the minimum set of services and the filter algorithms are applied. The results in Figure-7 show that the optimization algorithms shorten the execution time of the data mashup. This performance gain is due to the following factors: (*i*) the minimum set of services algorithm eliminates all redundant and irrelevant services whose invocations would return only redundant tuples on the mashup's outputs, and (*ii*) the filter algorithm eliminates superfluous calls to component services that would return empty (or error) results.

Note that the overhead introduced by the minimum set algorithm at the mashup creation time is justified by the performance gain found at the mashup execution time since mashups are created once and used many times.

5. Related Works

Since the data mashup research problem is relatively new, there has been only a small amount of research work addressing it. In the following, we review the most prominent ones of these works and compare them to ours.

The Web Service Management System (WSMS) in [14] allows users to mashup data services by directly expressing their queries in terms of data services' relations. Contrary to our system, the WSMS's users are assumed to have an understanding of the semantics of the services that are available to them. The work introduced two optimization algorithms. The first exploits the service selectivity to arrange component services in the mashup. Services with the lowest selectivity are executed first in the mashup to reduce the number of tuples that must be processed by subsequent services in the plan. The second uses variable chunk volumes for data exchanged among component services. Compared to WSMS, our system takes a more fundamental approach to the optimization issue by selecting only the minimum number of services that cover the mashup query. We use also the services constraints to filter out superfluous calls to services.

The Web Service Mediator System WSMED [13] allows users to mashup data services by defining relational views on top of them. Users can then query data by formulating their mashup queries over defined views. Users can also enhance defined views with primary-key constraints which can be exploited to optimize the mashups. The main drawback of the WSMED system is its high reliance on users; i.e. users are supposed to import the services relevant to their needs; define views on top of them and enhance the views with primary key constraints. The latter task requires from users to have a good understanding of the services' semantics. In our system, DaaS Web services are modeled as RDF views over domain ontologies where *primary key* constraints are defined explicitly by the concepts' skolem functions, thus the discussed *Primary key* based optimizations are included by default in our query processing model.

The CLIDE System [12] addresses the problem of semi-automatic interactive data mashup query formulation over a set of data services. The system helps mashup creators formulate feasible queries over data services by proposing a set of actions with which the query remains answerable. Once again, the data mashup creators are assumed to understand the semantics of exposed data services when they make actions during the mashup query formulation process. Furthermore, users are

supposed to drop code to wire the selected data services and do intermediate data processing operations (e.g. join, select, etc) among selected services. In addition, the work addresses only specific queries as opposed to parameterized queries addressed in this paper.

In other academic mashup systems [15,16,7,10], data mashup users are required to select the data services manually (which assumes they are able to understand their semantics), figure out the execution plan of selected services (i.e. the services *orchestration* in the mashup) and connect them to each other and drop code (in JavaScript) to mediate between incompatible inputs/outputs of involved services. This prevents average users from mashing up data services at large. Our mashup system addresses this limitation by proposing a declarative mashup approach, where users need only to focus on the required data and the system will find and mash-up the services for them.

A considerable body of recent work addresses the problem of composition (or orchestration) of multiple web services to carry out a particular task, e.g. [18,19]. In general, that work is targeted more toward workflow-oriented applications (e.g., the processing steps involved in fulfilling a purchase order), rather than applications coordinating data obtained from multiple web services, as addressed in this paper. Although these approaches have recognized the benefits of optimizing compositions, they have not, as far as we are aware, investigated the selection of minimum set of services or used the inputs/outputs data value constraints for the optimization.

6. Conclusion

In this paper we presented a query model to resolve parameterized queries over DaaS Web services. We proposed to rewrite mashup queries in terms of virtual RDF views representing meaningful data queries in a given application domain. We presented also a set of optimization algorithms to speed up the mashup execution time and evaluated their introduced performance gain. As a future work, we would like to extend our data mashup optimization framework with a mechanism that takes into account the quality of service QoS aspects (like the service response time, service reputation, etc).

7. References

[1] M. Barhamgi, D. Benslimane, and B. Medjahed, "A Query Rewriting Approach for Web Service Composition," *IEEE Transactions on Services Computing (TSC)*, vol. 3, no. 3, pp. 206-222, 2010.

[2] Business Process Execution Language for Web Services. ftp://www6.software.ibm.com/developer/wsbpel.pdf.

[3] D. Butler, "Mashups mix data into global service," *Nature*, vol. 439, pp. 6-7, January 2006., Available: http://dx.doi.org/10.1038/439006a.

[4] D. Calvanese, and M. Lenzerini, "Conjunctive Query Containment under Description Logics Constraints," *CoRR*, vol. 1, no. 2, pp. 9-30, 2005.

[5] Michael J. Carey, "Declarative Data Services: This Is Your Data on SOA," in *IEEE International Conference on Service-Oriented Computing and Applications*, California, USA, 2007, p. 4.

[6] A. Dan, R. Johnson, and A. Arsanjani, "Information as a Service: Modeling and Realization," in *International Conference on Software Engineering*, 2008, p. 2.

[7] H. Elmeleegy, A. Ivan, R. Akkiraju, and R. Goodwin, "Mashup Advisor: A Recommendation Tool for Mashup Development," in *2009 IEEE International Conference on Web Services (ICWS 2009)*, China, pp. 337-344.

[8] A. Halevy, A. Mendelzon, Y. Sagiv, and D. Srivastava, "Answering Queries Using Views," in *PODS*, 1995, pp. 95-104.

[9] Anant Jhingran, "Enterprise Information Mashups: Integrating Information, Simply," in *VLDB*, Seoul, Korea, 2007, pp. 3-4.

[10] Anne H. H. Ngu, Michael Pierre Carlson, and Hye-young Paik, "Semantic-Based Mashup of Composite Applications," *IEEE Transactions on Services Computing*, vol. 3, no. 1, pp. 2-15, 2010.

[11] Aris M. Ouksel, Oana Jurca and Karl Aberer, "Efficient Probabilistic Subsumption Checking for Content-Based Publish/Subscribe Systems," in *Middleware 2006, 7th International Middleware Conference*, Australia, 2006, pp. 121-140.

[12] M. Petropoulos, A. Deutsch, and Y. Katsis, "Exporting and interactively querying Web service-accessed sources: The CLIDE System," *ACM Trans. Database Syst.*, vol. 4, no. 32, 2007.

[13] Manivasakan Sabesan and Tore Risch, "Adaptive Parallelization of Queries over Dependent Web Service Calls," in*, WISS 2009*, China, 2009.

[14] Utkarsh Srivastava, Kamesh Munagala, and Jennifer Widom"Query Optimization over Web Services," in *VLDB*, Seoul, Korea, 2006, pp. 355-366.

[15] Junichi Tatemura et al., "UQBE: uncertain query by example for web service mashup," in *SIGMOD Conference*, Canada, 2008, pp. 1275-1280.

[16] Junichi Tatemura et al., "Mashup Feeds: : continuous queries over web services," in *SIGMOD Conference*, 2007, pp. 1128-1130.

[17] H. Truong and S. Dustdar, "On Analyzing and Specifying Concerns for Data as a Service," in *The 2009 Asia-Pacific Services Computing Conference (IEEE APSCC 2009)*, Singapore, 2009, pp. 7-11.

[18] M. Shiaa, J. Fladmark, and B. Thiell, "An Incremental Graph-based Approach to Automatic Service Composition" Proc. of the Int. Conf. on Services Computing (SCC'08), Honolulu, 2008.

[19] P. Hennig and W. Balke, "Highly Scalable Web Service Composition Using Binary Tree-Based Parallelization," Proc. of the Int. Conf. on Web Services (ICWS'10), Los Alamitos, USA, 2010.