A Data Integration Approach to Automatically Composing and Optimizing Web Services

Snehal Thakkar, José Luis Ambite and Craig A. Knoblock

University of Southern California/ Information Sciences Institute
4676 Admiralty Way
Marina Del Rey, CA 90292
USA
{thakkar, ambite, knoblock}@isi.edu

Abstract

In this paper we show how data integration techniques can be used to automatically compose new web services from existing web services. A key challenge is to optimize the execution of the composed web services. We introduce a novel technique termed tuple-level filtering that optimizes the execution of the composed web services by reducing the number of web service requests. Moreover, we combine the tuple-level filtering algorithm with a technique that includes additional web service requests in the composition in order to improve filtering and further optimize the execution. Our initial experimental evaluation shows that our optimization techniques can reduce the execution time of the composed web services by up to two orders of magnitude.

1. Introduction

A key promise of web services is seamless integration of information from various sources. The web services protocols (e.g., SOAP, WSDL) provide the infrastructure to address syntactical issues involved in integrating data from various data sources. Using this infrastructure, one can build new exciting applications on the web that integrate information from different web services and web sources. The true potential of web services can only be achieved if web services are used to create new web services that provide more functionality compared to existing services. Web service composition research has largely focused on producing specifications and tools to manually write web services, for example, using languages like BPEL4WS and WSFL, see [16] for a survey. Some initial research on automatic composition based on pre-defined HTN schemas is described in [20]. However, fully automatic web service composition in response to arbitrary user request remains an open problem.

Some commercial web services involve complex transactions and fully automatic web service composition may not be possible. However, it is possible to fully automate the composition of the large class of information-producing web services. In this paper, we describe how we build on existing mediator-based approaches to support the automatic composition of web services. In the context of automatically composing new web services from existing web services, the existing web services can be viewed as data sources. In recent years various mediator systems, such as the Information Manifold [12], InfoMaster [7], InfoSleuth [3], and Ariadne [9] have been used to provide a unified query interface to various data sources. At the same time the theoretical fundamentals of data integration have been investigated and are now well understood [10, 11]. The traditional mediator systems accept a specific user query and reformulate this query into a combination of source queries that can answer the specific user query.

Our mediator framework extends existing mediator-based approaches in two ways. First, the mediator-based systems typically provide an answer to a specific user query. We extend this by allowing a user to specify the class of queries the service should support. For example, the system may get a request to create a web service that accepts make of the car and price range and returns a list of cars made by the given maker and are for sale within the given price range.

Second, we describe a novel algorithm termed tuple-level filtering to optimize the integration plans for the composed web services. The tuple-level filtering algorithm utilizes the constraints in the source descriptions to reduce the number of requests sent to the existing web services. We also describe an extension to the tuple-level filtering algorithm to insert sensing operations in the integration plan to further optimize the execution of the composed web services.

The rest of the paper is organized as follows. Section 2 describes a motivating example that is used in the rest of the paper to clearly describe various concepts. Section 3 provides description of how the existing view integration algorithms can be utilized to compose web services. Section 4 presents the tuple-level filtering algorithm to optimize the execution of the composed web services. Section 5 provides the experimental evaluation of our approach. Section 6 describes the comparison of our approach with the existing research. Finally, Section 7 concludes the paper by discussing the contribution of the paper and directions for future work.

2. Motivating Example

In this section, we will describe some web services and queries that will be used in the rest of the paper to explain various concepts. As we introduced in [18], we can model the input/output behaviour of web services as data sources with binding pattern restrictions. Assume our mediator system has modelled the following web services as data sources.

CitytoCounty($city^b$, $state^b$, $county^f$)
LAProperty($address^b$, $city^f$, zip^f, $value^f$)
TNProperty($address^b$, $city^b$, $county^f$, zip^f, $value^f$)
YellowPages($name^b$, zip^f, $city^f$, $state^f$, $address^f$, $phone^f$)

The *CitytoCounty* source accepts a city and a state as an input and provides the county in which the city is located. The *LAProperty* web service accepts an address in "Los Angeles County" and provides the value of the property located at the given address. Similarly, the *TNProperty* web service accepts an address and city in the state of Tennessee and provides the property value and county information for the address. The *YellowPages* web service accepts a business name and provides the addresses for all the locations of the given business.

Our goal is to allow the users to compose and efficiently execute new web services using the above mentioned services. In [18], we described a mediator based approach to dynamically compose web services. While the approach described in [18] can compose web services, the composed web services plans may send unnecessary requests to the existing web services. In this paper, we describe an extension to optimize the plans generated in [18]. To clarify various concepts in this paper, we will use the request to create a web service that can "find property values for all locations of the given business". An example query for the newly composed web service is to "Find property values for all 'McDonalds' locations". In the next section, we briefly describe the procedure of creating new web services using the mediator based approach described in [18].

3. Mediator-based Web Service Composition

Recently there has been a lot of research on web service composition [1, 5, 14, 16, 18]. In [18], we described a mediator based approach to compose web services. The key intuition behind our approach was to utilize the techniques developed in the view integration research to automatically compose web services. Our approach to compose web services works in three steps. First, a domain expert designs a set of domain predicates and describes available web services as views over the domain predicates. For the example web services shown in Section 2 the domain expert may use the following domain predicates:

Cities(city, state, county)
BusinessProperties(name, address, city, county, state, zip, phone, value)

Traditionally, various mediator systems utilize either the Local-As-View approach [11] or the Global-As-View approach [6] to describe the relationship between domain predicates and available data sources. In the Global-As-View approach the domain predicates are described as views over available data sources. In the Local-As-View approach the data sources are described as views over the domain predicates. Adding additional data sources in the Local-As-View model is much easier compared to the Global-As-View model [11]. Therefore, our mediator system utilizes the Local-As-View model. For the given example, our mediator system describes the data sources as views over the domain predicates as follows:

R1: LAProperty(addr, city, zip, val):-
BusinessProperties(name, addr, city, county, state, zip, phone, val)^
county = 'Los Angeles'^ state = 'C A'
R2: TNProperty(addr, city, county, zip, value):-
BusinessProperties(name, addr, city, county, state, zip, phone, val)^
state = 'TN'
R3: YellowPages(name, zip, city, state, address, phone):-
BusinessProperties(name, address, city, county, state, zip, phone, value)
R4: CitytoCounty(city, state, county):-
Cities(city, state, county)
R5: CitytoCounty(city, state, county):-
BusinessProperties(name, address, city, county, state, zip, phone, value)

The source descriptions are given to the mediator. In addition to the source descriptions, we also provide mediator the rules about the functional dependencies in our domain model. The functional dependency relationships are provided by the domain expert. For example, in the domain model described in Section 2, the attributes *value* is functionally dependent on the attributes

address, city, and *state.* The rules to encode the functional dependency in the mediator are shown below.

FR1: equals(value, value') :-
BusinessProperties(name, addr, city, county, state, zip, phone, value)^
BusinessProperties(name', addr', city', county', state', zip', phone', value')^
equals(addr, addr')^
equals(city, city')^
equals(state, state')
FR2: equals(x, z) :- equals(x, y) ^ equals(y, z)

The rule FR1 states that if there exist two tuples in the relation *BusinessPriorities,* with the exact same values for the attributes *address, city,* and *state,* then the attribute *value* must have the same value in both tuples. The rule FR2 is inserted to ensure the transitivity property of the equality predicate.

Our mediator utilizes the Inverse Rules algorithm [4] to generate a datalog program for the new web service. The request to compose a web service that accepts a name of a business and returns property values for all the business locations can be formulated as the following query.

QR1: Q1(name, addr, city, st, zip, ph, val):-
BusinessProperties(name, addr, city, cty, st, zip, ph, val)^
name = <name>

The <name> denotes that the web service accepts a parameter termed *name.* Section 3.1 describes the process of generating a datalog program for the new web service using the domain model, the source descriptions, the functional dependency rules, and the query.

3.1 Inverse Rules algorithm

The Inverse Rules algorithm [4] is a query reformulation algorithm for the Local-As-View approach. There are also other query reformulation algorithms for the Local-As-View approach, such as, the Minicon algorithm [15]. In this paper we describe the query reformulation process using the Inverse Rules algorithm. However, the optimization algorithm described in this paper is applicable to any system that utilizes the Local-As-View model.

The first step of the Inverse Rules is to invert the source definitions to obtain definitions for all global relations as views over the source relations as ultimately only the requests on source relations can be executed. In order to generate the inverse view definition, the Inverse Rules algorithm analyzes all view definitions. For every view definition, $V(X) :- S_1(X_1),\ldots,S_n(X_n)$, where X and X_i refer to set of attributes in the corresponding view or relation, the Inverse Rules algorithm generates n inverse rules, for i = 1,..,n, $S_i(X'_i) :- V(X)$, where if $X_i \in X$, X'_i is the same as X_i else X_i is replaced by a function symbol [4]. For the given example, the Inverse Rules algorithm analyzes the view definitions and generates the following rules.

IR1: BusinessProperties(fnlap(), addr, city, 'Los Angeles', 'CA', zip, fplap(), val) :-
LAProperty(addr, city, zip, val)
IR2: BusinessProperties(fntnp(), addr, city, county, 'TN', zip, fptnp(), val) :-
TNProperty(addr, city, county, zip, val)
IR3: BusinessProperties(name, addr, city, fcyp(), state, zip, phone, fvyp()) :-
YellowPages(name, zip, city, state, addr, phone)
IR4: Cities(city, state, county):-
CitytoCounty(city, state, county)
IR5: BusinessProperties(fncc(), facc(), city, county, state, fzcc(), fpcc(), fvcc()) :-
CitytoCounty(city, state, county)

The rule IR1 is the result of inverting the rule R1 from the source descriptions. For clarity purposes, we have used a shorthand notation for the Skolem functions. In general the Skolem functions would have the rest of the attributes in the head as arguments. For example, Skolem function *fnlap()* would be written as *fnlap(addr, city, zip, val).* The rules IR2 to IR5 are result of inverting rules R2 to R5 from the source descriptions.

Given the source descriptions and the query, it is clear that the mediator can not evaluate the query without the functional dependencies. As all the rules to compute the predicate *BusinessProperties* result in function symbols either for *name* attribute or the *value* attribute. Therefore, the mediator must utilize the functional dependencies. As described in [4], for every attribute X in the head of the query, that participates in a functional dependency relationship the mediator replaces X with X' in the body of the query and inserts a new predicate equals(X, X') in the query. For the example query, attribute *val* participates in the functional dependency. Having added the equals predicate the resulting datalog program looks as follows:

Reasoning with the functional dependencies (see [4] for details), we can rewrite the program in Figure 1 to the equivalent program shown in Figure 2.

QR1: Q1(name, addr, city, st, zip, ph, val):-
BusinessProperties(name, addr, city, cty, st, zip, ph, val')^
name = <name> ^ equals(val, val')
FR1: equals(value, value') :-
BusinessProperties(name, addr, city, county, state, zip, phone, value)^
BusinessProperties(name', addr', city', county', state', zip', phone', value')^
equals(addr, addr')^

```
        equals(city, city')^
        equals(state, state')
FR2: equals(x, z) :- equals(x, y)^ equals(y, z)
IR1: BusinessProperties(fnlap(), addr, city, 'Los Angeles',
                'CA', zip, fplap(), val) :-
        LAProperty(addr, city,  zip, val)
IR2: BusinessProperties(fntnp(), addr, city, county, 'TN',
                zip, fptnp(), val) :-
        TNProperty(addr, city, county, zip, val)
IR3: BusinessProperties(name, addr, city, fcyp(), state,
                zip, phone, fvyp()) :-
        YellowPages(name, zip, city, state, addr, phone)
IR4: Cities(city, state, county):-
        CitytoCounty(city, state, county)
IR5: BusinessProperties(fncc(), facc(), city, county, state,
                fzcc(), fpcc(), fvcc()) :-
        CitytoCounty(city, state, county)
```

Figure 1 Datalog Program with Function Symbols

```
PR1: Q1(name, addr, city, cty, st, zip, ph, val):-
        YellowPages(name, zip, city, st, addr, ph)^
        name = <name>^
        LAProperty(addr, city, zip, val)
PR2: Q1(name, addr, city, cty, st, zip, ph, val):-
        YellowPages(name, zip, city, st, addr, ph)^
        name = <name>^
        TNProperty(addr, city, cty, zip, val)
```

Figure 2 Generated Datalog Program

The datalog program shown above can be used to host a web service that accepts the name of a business and provides property values for all business locations[1]. When the new web service receives a request to obtain property values for all locations of some business, e.g. 'McDonalds', it first obtains all locations of 'McDonald' from the *YellowPages* web service. Next, one request for each location is sent to the *LAProperty* and *TNProperty* sources to obtain the property values for the given locations. The results from both services are merged and returned to the user. While the above mentioned datalog program works well, it is not the most efficient program. We can optimize this program further by utilizing the source descriptions. The next section describes a tuple-level filtering technique that allows us to add filters to the plan to optimize the generated datalog program.

[1] As the mediator only has access to property tax services covering the state of Tennessee and the county of Los Angeles, the new web service will only provide property values for the business locations in the state of Tennessee or in the county of Los Angeles, which is maximally complete answer given the available services.

4. Optimizing web service composition plans by tuple-level filtering

In previous work [19], we introduced the idea of inserting filters based on source descriptions in data integration programs in order to reduce the number of requests sent to the data sources. In this paper, we generalize this technique and describe the conditions under which a mediator can optimize a web service composition represented by the datalog program generated by the Inverse Rules algorithm [4]. The key idea is to use the constraints in the source descriptions to add filters that eliminate provably useless calls to each web service. First, we describe the algorithm for filter introduction when the attributes needed to evaluate the filters already appear in the composition plan. Second, we extend the algorithm by including additional *sensing* web services that produce the attributes needed for the filters if they are not already present in the plan. In spite of including these additional sensing services, the resulting composition plans are often more cost-efficient since they take advantage of the discriminative power of the constraints.

Our algorithm for tuple-level filtering is shown in Figure 3. The algorithm takes as input the source (web service) descriptions and the datalog program generated by applying the inverse rules algorithm to the source descriptions. The output is an optimized datalog program. The algorithm represents an optimization search for the most cost-effective program that includes those constraints and sensing operation whose savings outweigh their evaluation costs. We denote each choice point by the keyword choice.

4.1 Introducing Filtering Constraints

The algorithm of Figure 3 optimizes each rule independently. Recall that each inverse rule has as head a domain predicate and as body a conjunctive query of source predicates (representing the web services) and constraints (equality and order). The algorithm first collects the attributess that are bound to constants in the body of the rule (line 3). Since the mediator knows the value of these attributes, it can evaluate constraints on them. Then, for each source (web service) predicate the algorithm retrieves its source description and for each constraint in the source description (lines 4-5), it checks whether it can evaluate the constraint *before* it calls the web service. The constraint can be evaluated if we know the values for the s involved in the constraint (line 6) at that point in the evaluation of the rule body (line 7). Intuitively, the web service will be called with input values that are part of a tuple that the body of the rule is constructing. If we know that the tuple is not relevant for that web service because it violates some constraint in the service description, it is better to filter such tuple and avoid the call to the service, which will certainly fail.

Tuple-level Filtering(SD, IR)
Input: SD: Source Descriptions (LAV rules)
IR: Inverse Rules
Output: Optimized Inverse Rules
Algorithm:
1. SP := heads of SD /* source predicates */
2. For each rule R in IR
3. B := attributes bound to constants in the body of R
4. For each source predicate S in body(R)
5. For each constraint C in the source description for S
6. A := attributes of C
7. If A $\subseteq$ B
8. Then /* insert filtering constraint */
9. **choice**[insert constraint C before S in body(R)]
10. Else /* insert sensing source predicate */
11. If $\exists$ source predicate SS in SP such that
12. inputs(SS) $\subseteq$ B and A $\in$ outputs(SS) and
13. $\exists$ functional dependency: inputs(SS) $\rightarrow$ A
14. Then **choice**[
15. insert predicate SS before S in body(R)
16. B := B $\cup$ attributes(SS)]
17. B := B $\cup$ attributes(S) /* update bound attributes */

Figure 3 Tuple-level Filtering Algorithm

Since the selectivity factor for a constraint may be small, it may not be worth including it in the composition plan. Thus, we make constraint inclusion a choice point in the algorithm (line 9). Only by estimating the expected evaluation cost, the mediator can be certain that the constraint inserted actually produces a more efficient plan.

To clarify the algorithm, consider our running example. The optimized datalog program generated after inserting constraints is shown in Figure 4. The source description for the *LAProperty* service contains order constraints on the attributes state and county. The constraint on the *state* attribute is used to add a filtering constraint to the rule PR1. The order constraint on the attribute *county* for the *LAProperty* service is (for the moment) ignored since no web service in the integration plan produces a value for the *county* attribute. Similarly, the order constraint on the *TNProperty* web service is used to add a filtering constraint to the rule PR2. As a result of the optimization, we obtain the new rules TR1 and TR2, as shown in Figure 4.

TR1: Q1(name, addr, city, cty, st, zip, ph, val):-
YellowPages(name, zip, city, st, addr, ph)^
name = <name>^
st = 'CA'^
LAProperty(addr, city, zip, val)
TR2: Q1(name, addr, city, cty, st, zip, ph, val):-
YellowPages(name, zip, city, st, addr, ph)^
name = <name>^
st = 'TN'^
TNProperty(addr, city, cty, zip, val)

Figure 4 . Tuple-level Filtering without Sensing

As a result of the filtering instead of sending one request for each tuple from the *YellowPages* web service to the *LAProperty* and the *TNProperty* web service, the optimized datalog program only sends tuples with attribute *state* equals "CA" to the *LAProperty* web service and tuples with *state* equals "TN" to the *TNProperty* web service. While the optimized datalog program reduces the number of requests sent to the property tax web services significantly, we may be able get more reduction by utilizing the order constraints that were ignored due to unavailability of some attributes. In the next section, we describe an extension to the tuple-level filtering algorithm to use sensing operations for further optimization.

4.2 Introducing Sensing Operations

The mediator system can further optimize a plan by adding sensing operations. So far, the mediator system only added filters for constraints whose attributes were already available in the plan, e.g., state. Nevertheless, the mediator can call other services to produce the attributes required by other constraints in the source descriptions. We call these additional web service calls *sensing* operations, since they are use to gather additional information that can be used to discriminate among the remaining service calls.

The introduction of sensing source (web service) predicates is show in lines 10 to 16 of the tuple-level filtering algorithm of Figure 3.

There are several conditions that a sensing web service predicate (SS) must satisfy in order to be a valid addition to the datalog program. First, the inputs of the web service (inputs(SS)) must already be computed by the composition plan[2]. Second, the desired constraint attribute (A) must be among the service outputs (outputs(SS)). Third, and most important, the service must satisfy a functional dependency between its inputs and the desired output attribute. This last condition is crucial to ensure that the semantics of the query does not change (nor the number of answers). Intuitively, the body of the rule is computing a tuple and the functional dependency restriction ensures that the tuple is only extended. Without the functional dependency the join with the new source may generate several tuples changing the semantics and results of the query.

We illustrate this reasoning with our running example. The optimized datalog program including sensing operations is shown in Figure 5. In particular, it includes in SR1 a call to the CitytoCounty web service as a sensing operation that produces the attribute *county*. Then the constraint on county can be introduced and enforced. In rule SR1 the mediator will check that the county of a

[2] Actually we could extend our algorithm to add sensing sources recursively, that is, a sensing source could provide inputs for another sensing source

business property is in fact "Los Angeles County" before sending a request to the *LAProperty* web service. As an example of the savings obtained by this technique, consider that if the *YellowPages* web service returned 150 tuples with *state* equals "CA" and only 25 of them were in Los Angeles County, the mediator would only send 25 requests to the *LAProperty* web service as opposed to 150 requests.

SR1: Q1(name, addr, city, cty, st, zip, ph, val):-
YellowPages(name, zip, city, st, addr, ph)^
name = <name>^ st = 'CA'^
CitytoCounty(city, st, cty)^
cty = 'Los Angeles'^
LAProperty(addr, city, zip, val)
SR2: Q1(name, addr, city, cty, st, zip, ph, val):-
YellowPages(name, zip, city, st, addr, ph)^
name = <name>^
st = 'TN'^
TNProperty(addr, city, cty, zip, val)

Figure 5 Tuple-level Filtering with Sensing

5. Experimental Results

In order to evaluate the saving provided by our algorithms we performed experiments for the motivating example using the following web services: (1) YellowPages web service[3], (2) Los Angeles county property tax web service[4], (3) Tennessee state property tax records web service[5], and (4) County information web service[6]. For the purpose of the experiments we converted the above mentioned web sites into web services using Fetch Agent platform[7]. The source descriptions were exactly the same as the motivating example.

We performed two experiments corresponding to two web service creation requests. In the first experiment, we send a request to create a web service that accepts name of a business and provides a list of all locations of the business and their values. In the second experiment, the request is to create a web service that accepts an address and provides the value of the property at the given address. All the experiments were run using a computer having Intel Pentium 4 processor operating at 2.2GHz and having 1 GB of memory.

The results of the first experiment are shown in Table 1. The input column of the Table 1 shows the different inputs that we passed to the composed web service. The Inverse rules algorithm without any optimizations would send all the business locations to both property tax web services. Thus, in the first example input 'Red Roof Inn', the program generated by the Inverse Rules algorithm sends 356 requests to the Los Angeles county property tax web service and the Tennessee state property tax web service, for a total of 713 requests. Therefore, the datalog program generated by the Inverse Rules program takes 8747 seconds.

The datalog program generated by the combination of the Inverse Rules algorithm and tuple-level filtering without sensing reduces the number of requests sent to each property tax web services considerably. In the first example, only 10 requests are sent to the Los Angeles county property tax web service and only 17 requests are sent to the Tennessee state property tax web service. Therefore, the execution time reduces to 92 seconds and only 28 requests are sent to different web services.

Finally, if the mediator adds sensing operations, the number of requests sent to the Los Angeles county property tax web service is further reduced to 4 requests. As there are 5 distinct cities with state equals "CA" in the result, the sensing operation requires sending 5 requests to the *CitytoCounty* web service. After the optimization with sensing only 27 total requests are sent to different web services. This is a huge improvement over the 713 requests that would have been sent by the unoptimized datalog program generated by the Inverse Rules algorithm without any optimization. This optimization results in the execution time of 88 seconds, which is about two orders of magnitude less than the execution time of the

Input	# tuples from YellowPages	# of tuples for state		# of tuples in county	Time in Seconds		
		Tennessee	California	Los Angeles	Inverse Rules	Tuple-level Filtering	Tuple-level Filtering With Sensing
Red Roof Inn	356	17	10	4	8747	92	88
Fetch	78	2	7	2	1578	72	67
Whiz-bang	11	1	1	0	402	32	30

Table 1 Experimental Results for Query 1

[3] http://www.switchboard.com
[4] http://www.lacountyassessor.com/extranet/default.asp
[5] http://170.142.31.248
[6] http://www.naco.org/
[7] http://www.fetch.com

unoptimized Inverse Rules algorithm[8]. Similar improvements are seen for other inputs.

For the second experiment, the mediator creates a web service that accepts an address from the user and finds the value of the property at the given address by sending requests to the property tax web services. The datalog program generated by the Inverse Rules algorithm sends one request each to both property tax web services. The datalog program generated by the tuple-level filtering algorithm without sensing, only sends one request to one of the property tax web services based on the given address. Therefore, there is a small improvement in the execution time. The datalog program generated by the tuple-level filtering with sensing results in one request to county information web service and one request to one of the two property tax web services, resulting in slight improvement over the Inverse Rules algorithm.

In general, the tuple-level filtering algorithm with or without sensing works well in any domain where one or more attributes can be used to filter out requests to different web services. We have identified the following example domains where the tuple-level filtering with or without sensing operations would result in great improvements in the execution time: (1) integrating classifieds information from different newspapers that cover different cities, (2) integrating automobile information from different manufacturers, and (3) integrating phone directories of different institutions.

6. Related Work

The work presented on this research is very closely related to research in several areas. First area of related work is research on mediator systems such as, the Information Manifold [12] InfoMaster [7], InfoSleuth [3], and Ariadne [9]. In our knowledge, no mediator systems have been utilized to automatically compose web services. Our work utilizes the Inverse Rules [4] algorithm, which is a part of the view integration research. The tuple-level filtering algorithm can be utilized with any system that utilizes Local-As-View [11] model, i.e. we could use Minicon algorithm [15] or Bucket algorithm [11] instead of the Inverse Rules [4] algorithm.

Second area of related work is on optimization of data integration plans. In [8], the authors describe strategies to optimize the recursive and non-recursive datalog programs generated by the Inverse Rules algorithm. The research focus of their work is to remove redundant and to order access to different sources to reduce the query execution time in presence of overlapping data sources. Our optimization algorithm optimizes a plan for parameterized query as opposed to a specific query. Moreover, our optimization algorithm may insert sensing operations to optimize the query. Both approaches are complimentary as tuple-level filtering can be used to generate the initial datalog program for the composed web service and when the composed web service receives a request from the user algorithms from [8] can be used to optimize for the specific query. In [13], the authors describe a more efficient approach compared to [4] to handle binding pattern restrictions. In this paper we utilized the approach described in [4]. However, our optimization algorithms would work with the binding pattern satisfaction algorithm described in [13] as well.

There has been some research on automatic web service composition. In particular [20], describe a SHOP2 based system to automatically compose web services. In addition to the input and output constraints, their system can also handle web services with preconditions and effects. In [14], the authors use Golog templates to compose different web services. While this representation is powerful and can handle web services with preconditions and effects, their system requires a human to write different plan templates before the system can answer different user queries. The focus of both [14, 20] is on composing web services. Therefore, they do not address the issue of optimizing the execution of composed web services.

The idea of inserting filtering based on source descriptions was discussed in [19]. However, the mediator system in that paper utilized forward chaining to generate the integration plan. The resulting mediator system was unable to handle recursion or utilize functional dependencies. In this paper, we use the Inverse Rules algorithm to generate the integration plan to address those limitations. Similarly, we described the idea of using sensing operations to optimize data integration plans in [2]. In this paper, we have generalized the idea of using the sensing operations by utilizing the source descriptions and the generated integration plan to insert the sensing operations.

7. Discussion and Future Work

In this paper we have described a mediator based approach for automatic web service composition. We show that integration plans generated for new web services using the existing view integration algorithms are often inefficient. We described a novel algorithm termed tuple-level filtering algorithm that inserts sensing operations in the composed web services to further optimize the execution of the composed web services. We showed that using optimization algorithms described in this paper, we can reduce the execution time of the composed web services by up to two orders of magnitude.

The work described in this paper is an important step towards our goal of completely automatic composition of information gathering web services. Our next step is to

[8] It is easy to see that as the number of tuples returned from the YellowPages increases, the sensing operation would provide more improvements. For the final version we will have more experiments to prove the usefulness of adding the sensing operations.

apply our mediator to compose web services listed in a web service directory, such as UDDI. The key new challenge would be to automatically model web services in the directory as data sources in the mediator's domain model. Furthermore, we are looking to extend our mediator to support a wide variety of operations on heterogeneous data. For example, when integrating data from several data sources, a key challenge is to consolidate data from various data sources. We are working on incorporating an object consolidation system termed Active Atlas [17] in the mediator to address this challenge.

Acknowledgements

This material is based upon work supported in part by the National Science Foundation, under Award No. IIS-0324955, in part by the Air Force Office of Scientific Research under grant number F49620-01-1-0053, and in part by a gift from the Microsoft Corporation. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

References

1. Andrews, T., F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, *Business Process Execution Language for Web Services*. 2002.
2. Ashish, N., C.A. Knoblock, and A. Levy. *Information Gathering Plans with Sensing Actions*. in *European Conference on Planning, ECP-97*. 1997. Toulouse, France.
3. Bayardo Jr., R.J., W. Bohrer, R. Brice, A. Cichocki, J. Flower, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. *Infosleuth: Agent-based semantic integration of information in open and dynamic environments*. in *In Proceedings of ACM SIGMOD-97*. 1997.
4. Duschka, O.M., *Query Planning and Optimization in Information Integration*, in *Ph.D. Thesis, Computer Science*. 1997, Stanford University.
5. Florescu, D., A. Grünhagen, and D. Kossmann. *XL: an XML programming language for web service specification and composition*. in *Proceedings of the eleventh international conference on World Wide Web*. 2002.
6. Garcia-Molina, H., J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. *Integrating and Accessing Heterogeneous Information Sources in TSIMMIS*. in *Proceedings of the AAAI Symposium on Information Gathering*. 1995. Stanford, CA.
7. Genesereth, M.R., A.M. Keller, and O.M. Duschka. *InfoMaster: An information integration system*. in *In Proceedings of ACM SIGMOD-97*. 1997.
8. Kambhampati, S., E. Lambrecht, U. Nambiar, Z. Nie, and S. Gnanaprakasam, *Optimizing Recursive Information Gathering Plans in EMERAC*. To appear in Journal of Intelligent Information Systems, 2003.
9. Knoblock, C., S. Minton, J.L. Ambite, N. Ashish, I. Muslea, A. Philpot, and S. Tejada, *The ARIADNE Approach to Web-Based Information Integration.* International Journal on Intelligent Cooperative Information Systems (IJCIS), 2001. **10**(1-2): p. 145-169.
10. Lenzerini, M. *Data integration: A theoretical perspective*. in *In Proceedings of ACM Symposium on Principles of Database Systems*. 2002. Madison, Winsconsin, USA.
11. Levy, A., *Logic-Based Techniques in Data Integration*, in *Logic Based Artificial Intelligence*, J. Minker, Editor. 2000, Kluwer Publishers.
12. Levy, A.Y., A. Rajaraman, and J.J. Ordille. *Query-answering algorithms for information agents*. in *In Proceedings of AAAI-96*. 1996.
13. Li, C., *Computing Complete Answers to Queries in the Presence of Limited Access Patterns.* The VLDB Journal, 2003. **12**: p. 211-227.
14. McIlraith, S. and T.C. Son. *Adapting golog for composition of semantic web services*. in *Proceedings of the 8th International Conference on Knowledge Representation and Reasoning (KR '02)*. 2002. Toulouse, France.
15. Pottinger, R. and A. Levy, *A Scalable Algorithm for Answering Queries Using Views.* VLDB Journal, 2000: p. 484-495.
16. Srivastava, B. and J. Koehler. *Web Service Composition - Current Solutions and Open Problems*. in *In the Proceedings of the ICAPS workshop on Planning for Web Services*. 2003.
17. Tejada, S., C.A. Knoblock, and S. Minton, *Learning Object Identification Rules for Information Integration.* Information Systems, 2001. **26**(8).
18. Thakkar, S., J.L. Ambite, and C.A. Knoblock. *A view integration approach to dynamic composition of web services*. in *In Proceedings of 2003 ICAPS Workshop on Planning for Web Services*. 2003. Trento, Italy.
19. Thakkar, S., C.A. Knoblock, J.L. Ambite, and C. Shahabi. *Dynamically Composing Web Services from On-line Sources*. in *In Proceeding of 2002 AAAI Workshop on Intelligent Service Integration*. 2002. Edmonton, Alberta, Canada.
20. Wu, D., B. Parsia, E. Sirin, J. Hendler, and D. Nau. *Automating DAML-S Web Services Composition Using SHOP2*. in *2nd International Semantic Web Conference (ISWC2003)*. 2003.