The VLDB Journal (2005) 14(3): 330–353
DOI 10.1007/s00778-005-0158-4

REGULAR PAPER

Snehal Thakkar · José Luis Ambite · Craig A. Knoblock

Composing, optimizing, and executing plans for bioinformatics web services

Received: 7 July 2005 / Accepted: 7 June 2005 / Published online: 26 September 2005

Abstract The emergence of a large number of bioinformatics datasets on the Internet has resulted in the need for flexible and efficient approaches to integrate information from multiple bioinformatics data sources and services. In this paper, we present our approach to automatically generate composition plans for web services, optimize the composition plans, and execute these plans efficiently. While data integration techniques have been applied to the bioinformatics domain, the focus has been on answering specific user queries. In contrast, we focus on automatically generating *parameterized* integration plans that can be hosted as web services that respond to a range of inputs. In addition, we present two novel techniques that improve the execution time of the generated plans by reducing the number of requests to the existing data sources and by executing the generated plan more efficiently. The first optimization technique, called tuple-level filtering, analyzes the source/service descriptions in order to automatically insert filtering conditions in the composition plans that result in fewer requests to the component web services. To ensure that the filtering conditions can be evaluated, this technique may include sensing operations in the integration plan. The savings due to filtering significantly exceed the cost of the sensing operations. The second optimization technique consists in mapping the integration plans into programs that can be executed by a dataflow-style, streaming execution engine. We use real-world bioinformatics web services to show experimentally that (1) our automatic composition techniques can efficiently generate parameterized plans that integrate data from large numbers of existing services and (2) our optimization techniques can significantly reduce the response time of the generated integration plans.

Keywords Bioinformatics · Web service composition · Data integration · Query optimization · Dataflow-style streaming execution

S. Thakkar (✉) · J. L. Ambite · C. A. Knoblock
Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, CA 90292, USA
E-mail: thakkar@isi.edu

1 Introduction

There exist a large number of bioinformatics datasets on the web in various formats. There is a need for flexible and efficient approaches to integrate information from these datasets. Unlike other domains, the bioinformatics domain has embraced web standards, such as XML and web services. A web service is a program that can be executed on a remote machine using standard protocols, such as WSDL and SOAP. There exists a large number of bioinformatics data sources that are either accessible as web services or provide data using XML. For the bioinformatics data sources that provide their data as semi-structured web or text documents, we can use wrapper-based techniques [1–3] to access the data. Most of the available bioinformatics web services are information-providing services, i.e. these services do not change the state of the world in any way. For example, when a user queries the UniProt[1] website for details of a protein, the user provides a *uniprotid* and gets back the information about the protein. Sending this request does not result in side effects, such as charges to the user's credit card. The emergence of the large number of information-providing services has highlighted the need for a framework to integrate information from the available data sources and services.

In this paper, we describe our approach to automatically compose integration plans to create new information-providing web services from existing web services. When our framework receives a request to create a new web service, it generates a parameterized integration plan that accepts the values of the input parameters, retrieves and integrates information from relevant web services, and returns the results to the user. The parameterized integration plan is then hosted as a new web service. The values of the input parameters are not known at composition time. Therefore, the parameterized integration plan must be able to handle different values of input parameters. This is the key challenge in composing plans for a new web service. To further clarify this, consider the example shown in Fig. 1. We have access

[1] http://www.pir.uniprot.org/.

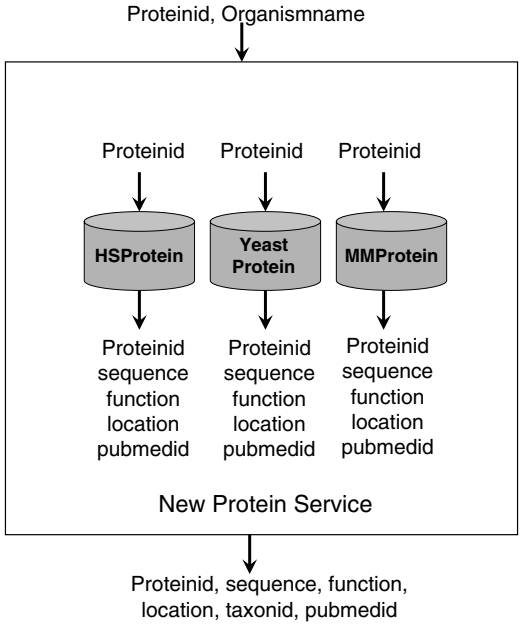

Fig. 1 Example composed service

to three web services, each providing protein information for different organisms. We would like to create a new web service that accepts the name of an organism and the id of a protein and returns the protein information from the relevant web service. Given specific values of the input parameters, traditional data integration systems can decide which web service should be queried. However, without knowing the values of the parameters, the traditional integration systems would generate a plan that requires querying all three web services for each request.

The key contribution of our approach is to extend the existing techniques to generate parameterized integration plans that can answer requests with different sets of values for the input parameters. This is similar to the problem of generating universal plans [4] in that the generated plan must return an answer for *any* combination of valid input parameters.

A key issue when generating parameterized plans is to optimize the plans to reduce the number of requests sent to the existing data sources. The existing optimization techniques utilize the constants in the user query to filter out unnecessary source requests and/or reorder the joins to produce more efficient plans. However, as we show with a detailed example later in the paper, those techniques are not enough when we apply them to the task of optimizing parameterized integration plans. Intuitively, we can improve the performance of the parameterized plans for the composed web services using two approaches: (1) by reducing the number of requests sent to web services and (2) by executing requests to the existing web services more efficiently. To that end, we describe two optimizations to reduce the response time of the composed web services: (1) a tuple-level filtering algorithm that optimizes the parameterized integration plans by adding filters based on the source descriptions of the existing web services to reduce the number requests made to the existing web services and (2) an algorithm to map the parameterized integration plans into dataflow-style, streaming execution plans that can be executed efficiently using a highly parallelized, streaming execution engine.

This paper builds on our earlier work, which presented preliminary results on tuple-level filtering [5, 6] and mapping datalog into streaming, dataflow-style execution system [7]. This article describes these techniques in more detail, shows how they can be applied to the bioinformatics domain, and contains new experimental results on real-world bioinformatics web services.

We begin by describing a motivating example that we use throughout the paper to provide a detailed explanation of various concepts. Next, we discuss how existing data integration techniques can be extended to model web sources as data sources and reformulate web service creation requests into parameterized integration plans. Next, we describe an optimization technique termed tuple-level filtering that introduces filters and sensing operations in the parameterized integration plan to reduce the number of requests to the existing web services. In addition, we present a discussion on the applicability of the tuple-level filtering in the bioinformatics domain. Then, we describe techniques to translate recursive and non-recursive datalog composition plans into integration plans that can be executed by a dataflow-style execution engine. Our experimental evaluation shows that the techniques described in this paper achieve a significant reduction in the response time of the composed web services. We conclude the paper by discussing the related work, contributions of the paper, and future work.

2 Motivating example

In this section, we describe a set of available web services and an example web service that we would like to create by composing the available services. The existing web services provide information about various proteins and interactions between different proteins. We model each web service operation as a data source with binding restrictions. The '$' before the attribute denotes that the value for the attribute is required to obtain the rest of the information, i.e., the attribute is a required input to the web service operation. Each data source provides information about one or more domain concept(s). A domain concept refers to a type of entity, e.g. protein.

As shown in Table 1, we have access to eight different web services that provide information about various proteins. Six of these web services, namely, *HSProtein*, *MMProtein*, *MembraneProtein*, *TransducerProtein*, *DIPProtein*, and *ProteinLocations* provide information about proteins. The *HSProteinInteractions* and *MMProteinInteractions* services provide information about interactions between proteins.

The *HSProtein*, *MMProtein*, *MembraneProtein* and *TransducerProtein* services accept the id of a protein and provide the name of the protein, the location of the protein

Table 1 Available web services

Concept	Source
Protein	HSProtein($id, name, location, function, sequence, pubmedid) MMProtein($id, name, location, function, sequence, pubmedid) MembraneProtein($id, name, taxonid, function, sequence, pubmedid) TransducerProtein($id, name, taxonid, location, sequence, pubmedid) DIPProtein($id, name, function, location, taxonid) ProteinLocations($id, $name, location)
Protein–protein interactions	HSProteinInteractions($fromid, toid, source, verified) MMProteinInteractions($fromid, toid, source, verified)

in a cell, the function of the protein, the sequence of the protein, and a pointer to articles that may provide more information about the protein.[2] The protein information services cover different sets of proteins. The *HSProtein* web service only provides information about human proteins, while the *MMProtein* web service provides information about mouse proteins. The *MembraneProtein* web service provides information about proteins located in the membrane, while the *TransducerProtein* provides information about all the proteins that act as transducers. The *DIPProtein* web service accepts a *proteinid* and provides *name*, *function*, *location*, and *taxonid* information for all proteins. The *ProteinLocations* service accepts a *proteinid* and *name* of the protein and provides the location of the protein.[3]

Similarly, we also have access to two web services that provide information about interactions between different proteins. Both web services accept a *proteinid* and provide ids of the interacting proteins, sources of the interaction, and information on whether the interaction was verified. The *HSProteinInteractions* gives information about human protein–protein interactions, while the *MMProteinInteractions* provides information about mouse protein–protein interactions.

Figure 2 shows the graphical representation of the relationships between the data sources and domain concepts. The square block in the figure (e.g. *Protein*) represents a domain entity. The diamond-shaped box (e.g. *ProteinProteinInteractions*) represents a relationship between domain entities. Cylindrical shapes denote sources. The dotted lines show the relationships between the sources and domain entities.

Given these sources, a user may want to create a new service by combining information from various sources.

[2] Since we are using a relational schema, we can only have one value for the pubmedid attribute. For simplicity, we assume that value for the pubmedid attribute is a URL that points to a page containing list of articles that refer to the protein.

[3] For simplicity, we assume that all sources utilize the same *proteinid* to identify proteins. If the available sources do not share common keys, we can use record linkage techniques, such as [8], to materialize a source that provides mapping between the keys of different sources.

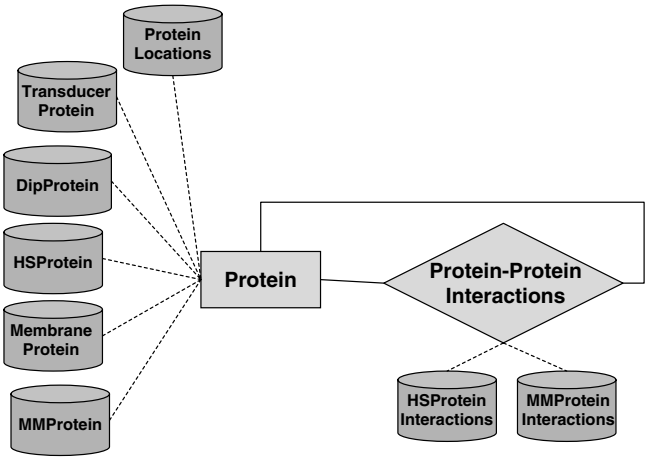

Fig. 2 Relationships between domain concepts and data sources

One such example is to create service that accepts a *proteinid*, queries relevant protein sources to obtain information about the protein, and returns the information to the user. The framework described in this paper allows the users to quickly create such web services. We would like to allow users to specify web service creation requests using the domain concepts. Our framework must generate an integration plan that determines relevant sources based on values of different input parameters.

As the motivating example for the rest of the paper, we would like our framework to create the service shown in Fig. 3 that accepts a *proteinid* and finds the sequence for the given protein and the id and sequence information (*toproteinid* and *toseq* attributes in the figure) about all the proteins with which it interacts either directly or indirectly. We use rounded rectangles to denote domain concepts. For example, the rounded rectangles with the *Protein* and *ProteinProteinInteractions* text denote retrieval operations from the *Protein* and *ProteinProteinInteractions* domain relations, respectively. Directed arrows in the figure denote a dependency between the two symbols connected by an arrow. For example, the *Join* operation cannot be performed until data is obtained from both *Protein* operations. In this example,

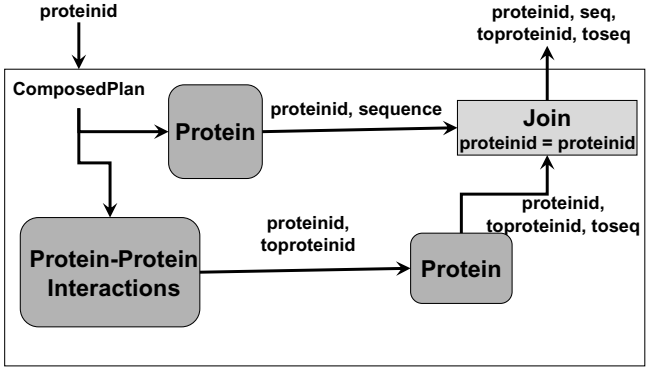

Fig. 3 Example of the integration plan of a desired web service

Protein and *ProteinProteinInteractions* are virtual relations. The task of our framework is to generate an integration plan that accepts the values for the input parameters, retrieves necessary information from the relevant source web services (e.g. *HSProtein*), and returns the response to the user.

3 Adapting data integration techniques to web service composition

In this section we describe an extension to the existing data integration techniques to solve the problem of generating parameterized integration plan for new bioinformatics web services. Most Life Sciences web services are information-providing services. We can treat information-providing services as data sources with binding restrictions. Data integration systems [9–12] require a set of domain relations, a set of source relations, and a set of rules that define the relationships between the source relations and the domain relations.

In Sect. 3.1 we describe how we can create a domain model for the given example. In Sect. 3.2 we describe how we use an existing query reformulation technique called Inverse Rules [13] to generate a datalog program to answer specific user queries. In Sect. 3.3 we describe our extensions to the existing data integration techniques to support the generation of parameterized integration plans for web service composition.

3.1 A4. Modeling web services as data sources

In order to utilize the existing web services as data sources, we need to model them as available data sources and create rules to relate the existing web services with various concepts in the domain. Typically, a domain expert consults the users and determines a set of domain relations. The users form their queries on the domain relations. For the example in Sect. 2, we have two domain relations with the following attributes:

Protein(id, *name*, *location*, *function*, *sequence*, *pubmedid*, *taxonid*)
ProteinProteinInteractions(*fromid*, *toid*, *taxonid*, *source*, *verified*)

The *Protein* relation provides information about different proteins. The *ProteinProteinInteractions* relation contains interactions between different proteins. As the *id* attribute in the *Protein* relation is the primary key, all other attributes in the *Protein* relation functionally depend on the *id* attribute. For the *ProteinProteinInteractions* domain relation, the combination of *fromid* and *toid* forms a primary key.

Once we have determined the domain relations, we need to define the relationships between the domain relations and the available web services. Traditionally, various mediator systems utilize either the Local-As-View approach [14], the Global-As-View approach [15], or the Global-Local-As-View (GLAV) [16] to describe the relationship between the domain predicates and available data sources. In the Global-As-View approach, the domain relations are described as views over available data sources. In the Local-As-View approach, the data sources are described as views over the domain relations. Adding data sources in the Local-As-View model is much easier compared to the Global-As-View model. Therefore, our data integration system utilizes the Local-As-View model. We define the data sources as views over the domain relations as shown in Fig. 4. The source descriptions (*SD1–SD8*) contain a source relation as the head of the rule and a conjunction of domain relations and equality or order constraints in the body of the rule.

SD1: HSProtein(id, name, location, function, sequence, pubmedid):-
Protein(id, name, location, function, sequence, pubmedid, taxonid) ∧
taxonid=9606

SD2: MMProtein(id, name, location, function, sequence, pubmedid):-
Protein(id, name, location, function, sequence, pubmedid, taxonid) ∧,
taxonid=10090

SD3: MembraneProtein(id, name, taxonid, function, sequence, pubmedid):-
Protein(id, name, location, function, sequence, pubmedid, taxonid) ∧
location='Membrane'

SD4: TransducerProtein(id, name, taxonid, location, sequence, pubmedid):-
Protein(id, name, location, function, sequence, pubmedid, taxonid) ∧
function='Transducer'

SD5: DIPProtein(id, name, function, location, taxonid):-
Protein(id, name, location, function, sequence, pubmedid, taxonid)

SD6: ProteinLocations(id, name, location):-
Protein(id, name, location, function, sequence, pubmedid, taxonid)

SD7: HSProteinInteractions(fromid, toid, source, verified):-
ProteinProteinInteractions(fromid, toid, taxonid, source, verified) ∧
taxonid=9606

SD8: MMProteinInteractions(fromid, toid, source, verified):-
ProteinProteinInteractions(fromid, toid, taxonid, source, verified) ∧
taxonid=10090

DR: ProteinProteinInteractions(fromid, toid, taxonid, source, verified):-
ProteinProteinInteractions(fromid, itoid, taxonid, source, verified) ∧
ProteinProteinInteractions(itoid, toid, taxonid, source, verified)

Fig. 4 Source descriptions and domain rule

In addition to the source descriptions, we also include the recursive domain rule *DR* to ensure that the *ProteinProteinInteractions* relation actually represents all protein–protein interactions, not just direct protein–protein interactions. A domain rule must contain exactly one domain relation as the head of the rule and a conjunction of domain relations, source relations, and equality or order constraints in the body of the rule. In general, we assume that we have the correct model for all available data sources and the data sources do not report incorrect data. However, our framework can handle incomplete data sources. For example, a web service that provides information about human proteins may only provide information about some human proteins.

Having defined the domain model and source descriptions, the users can send queries to the data integration

Q1:Q1(fromid, fromname, fromseq, frompubid, toid, toname, toseq, topubid):-
Protein(fromid, fromname, loc1, func1, fromseq, frompubid, taxonid) ∧
ProteinProteinInteractions(fromid, toid, taxonid, source, verified) ∧
Protein(toid, toname, loc2, func2, toseq, topubid, taxonid) ∧
taxonid = 9606 ∧
fromid = 19456

Fig. 5 Example query

system. Figure 5 shows an example query that asks the system to find information about the proteins with *proteinid* equal to '19456' and *taxonid* equal to '9606', and their interactions.

3.2 A5 Answering individual user queries

When a traditional data integration system gets a user query, it utilizes a query reformulation algorithm to generate a datalog program to answer the user query using the source descriptions, domain rules, and the user query. Our mediator is based on the Inverse Rules [13] query reformulation algorithm for the Local-As-View approach.

The first step of the Inverse Rules is to invert the source definitions to obtain definitions for all domain relations as views over the source relations as ultimately only the requests on the source relations can be executed. In order to generate the inverse view definition, the Inverse Rules algorithm analyzes all source descriptions. The rules *IR1* through *IR8* are the result of inverting the rules *SD1* through *SD8* from Fig. 4. The head of the rule *IR5* contains function symbols as the attributes *sequence* and *pubmedid* are not present in the source *DIPProtein*. For clarity purposes, we have used a shorthand notation for these Skolem functions. In general, the Skolem functions would have the rest of the attributes in the head of the view as arguments. For example, Skolem function *f6*(. . .) in rule *IR5* stands for *f6(id, name, function, location, taxonid)*.

Next, the mediator combines the domain rule, the relevant inverted rules shown in Fig. 6, and the user query shown in Fig. 5 to generate a datalog program to answer the user query. Figure 7 shows a graphical representation of the datalog program. We can use any datalog evaluation engine (as long as the datalog engine can retrieve data from remote sources and web services) to execute the program and get the answer to the user query. Given a *Proteinid*, the integration plan proceeds as follows. The given *Proteinid* is used to send requests to the three relevant protein information data sources. Note that the source *MMProtein* is not used as it has the constraint, *taxonid* = *10090*, which conflicts with a constraint in the user query. In addition, a request is sent to the *HSProteinInteractions* data source to obtain all interactions between the given protein and other proteins. The *MMProteinInteractions* data source is not used as it has a constraint on the attribute *taxonid* that conflicts with a constraint in the query. Next, the data integration system sends requests to the three relevant protein sources to find information about all the directly interacting proteins. The information about the given protein and interacting proteins is joined and provided as part of the output, while the ids of the interacting

IR1:Protein(id, name, location, function, sequence, pubmedid, f1(...)):-
HSProtein(id, name, location, function, sequence, pubmedid)

IR2:Protein(id, name, location, function, sequence, pubmedid, f2(...)):-
MMProtein(id, name, location, function, sequence, pubmedid)

IR3:Protein(id, name, f3(...), function, sequence, pubmedid, taxonid):-
MembraneProtein(id, name, taxonid, function, sequence, pubmedid)

IR4:Protein(id, name, location, f5(...), sequence, pubmedid, taxonid):-
TransducerProtein(id, name, taxonid, location, sequence, pubmedid)

IR5:Protein(id, name, location, function, f6(...), f7(...), taxonid):-
DIPProtein(id, name, function, location, taxonid)

IR6:Protein(id, name, location, f8(...), f9(...), f10(...), f11(...)):-
ProteinLocations(id, name, location)

IR7:ProteinProteinInteractions(fromid, toid, taxonid, source, verified):-
HSProteinInteractions(fromid, toid, source, verified)

IR8:ProteinProteinInteractions(fromid, toid, taxonid, source, verified):-
MMProteinInteractions(fromid, toid, source, verified)

Fig. 6 Automatically generated inverse rules

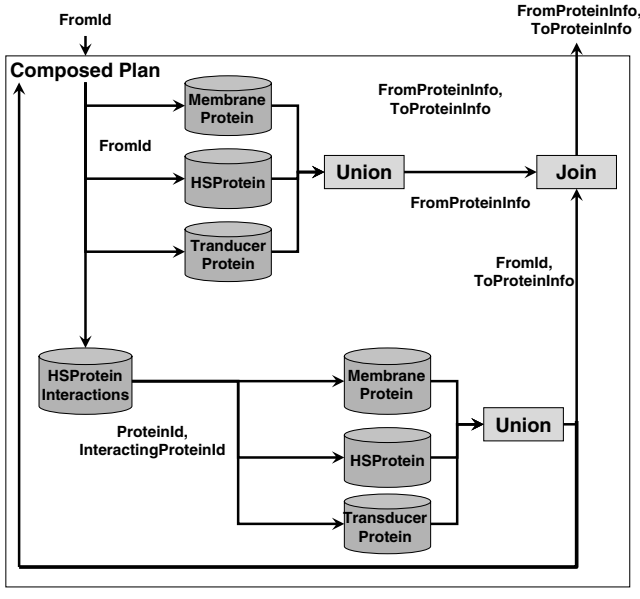

Fig. 7 Generated integration plan to answer the user query

proteins are used as input to the next iteration to obtain indirect interactions.

3.3 A6. Generating parameterized integration plans for web service composition

While data integration systems can be used to answer the user queries by integrating data from various data sources, the user still needs to specify the query for each request and needs to know the domain model. Ideally, we would like to create a web service that accepts some input parameters such as *proteinid*, executes a pre-determined program, and provides the results of the program to the user. In other words, we would like a data integration system to generate a web

```
Q1:Q1(fromid, fromname, fromseq, frompubid, toid, toname, toseq, topubid):-
      Protein(fromid, fromname, loc1, func1, fromseq, frompubid, taxonid),
      ProteinProteinInteractions(fromid, toid, taxonid, source, verified),
      Protein(toid, toname, loc2, func2, toseq, topubid, taxonid),
      (taxonid = !taxonid),
      (fromid = !proteinid)
```

Fig. 8 Parameterized query

service that a user can utilize over and over with different values for the inputs. A key difference between generating a web service and answering specific queries is that the data integration system needs to generate a parameterized integration plan that works for different values of the input parameters.

We extend the techniques described in Sect. 3.2 in two ways in order to automatically generate parameterized integration plans. First, instead of passing in specific queries to the data integration system, we pass in parameterized queries, such as the query shown in Fig. 8. We use the '!' prefix to denote a parameter. Unlike the specific query, the value of the parameter is not known to the mediator. The generated integration plan should accept *proteinid* and *taxonid* parameters. The arguments in the head of the query show the output of the generated plan. The generated plan should output the following attributes: *fromid*, *fromname*, *fromseq*, *frompubid*, *toid*, *toname*, *toseq*, *topubid*. The body of the datalog rule indicates the information that the generated plan would need to gather. For the given query, the generated plan should query the *Protein* relation to obtain *name*, *seq*, and *pubid* information for the given *proteinid*. Next, it should query the *ProteinProteinInteractions* relation to find all proteins that interact with the given protein. Finally, it should find the *name, seq*, and *pubid* information for all the interacting proteins. The information about the given protein and all the interacting proteins should be returned to the user.

Second, we modify the Inverse Rules [13] to treat the parameterized constraints in the query as runtime variables [17] since the value of the parameters is not known. Like the data integration system described in Sect. 3.2, our extended integration system also requires a domain model and source descriptions. To generate the parameterized integration plan, the mediator utilizes the Inverse Rules [13] technique. As the constraints in the query have parameters, it is not possible to filter out sources by checking for conflicting constraints. For example, even though there is a constraint on the *taxonid* attribute in the query and a constraint on the *taxonid* attribute in the description of the source *HSProtein*, as we do not know the value of the parameter *!taxonid*, we cannot exclude the *HSProtein* source from the generated plan. Instead, our system must utilize all available data sources for every domain relation. For the given query, the integration system needs to send requests to all four data sources to obtain information about proteins. Moreover, the integration system must also send requests to both protein–protein interactions data sources as shown in the integration plan in Fig. 9.

Once the integration system generates the parameterized integration plan, it can be hosted as a web service and the

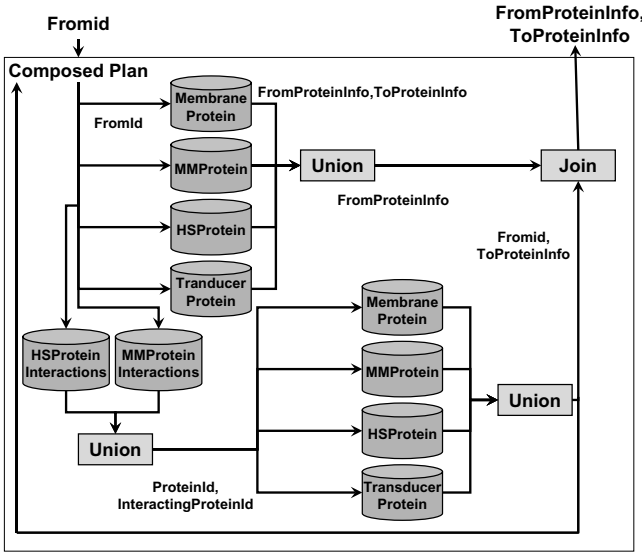

Fig. 9 Parameterized integration plan

users can query the web service by providing different values of *taxonid* and *proteinid*.

One advantage of our approach is that once a web service is composed using our framework, the users of the composed web service do not need to know the details of the mediator's domain model. As long as the users know what function the web service performs, they can use the service by providing the parameter values.

4 Optimizing web service composition plans using tuple-level filtering

While the generated integration plan can be hosted as a web service that provides complete answers for any given values of the input, it may send a large number of requests to the existing web services. This may result in slow response times for the composed web service. Therefore, it is important to optimize the generated integration plans to remove unnecessary requests to the component web services. For example, the user can send requests to the example web service described in Sect. 2 with different values of *proteinid*. However, each request to the composed web service may require a large number of requests to the composed web services. For example, when we invoke the composed service with '19456' as the value for the *proteinid* parameter, the composed service would need to call all the web services that provide protein information once for the given protein and once for each interacting protein.

There has been much work in the data integration community on the issue of reducing the response time of integration plans by removing redundant calls to data sources and ordering data accesses [18–20]. However, those optimizations are geared toward answering specific queries, while web service composition requires integration plans that can answer the parameterized queries. It may not be possible to

identify redundant or unnecessary calls to data sources in a parameterized integration plan until the execution time, when the parameter values are known. The existing optimization techniques rely on comparing constraints in the query with the constraints in the source descriptions to determine if a source may provide useful tuples to answer the user query. However, in case of the parameterized plans, the values of the input parameters participating in the constraints are not known at composition time. Therefore, the existing optimization techniques would not be able to remove any source requests from the composed parameterized plans, such as the one shown in Fig. 9. In this section, we describe a novel optimization algorithm termed tuple-level filtering that addresses this problem by optimizing the generic integration plans using the equality and order constraints in the source descriptions.

The key idea behind the tuple-level filtering algorithm is to use the equality (e.g., $x = 5$) and order constraints (e.g., $x < 5$) in the source descriptions to add filters that eliminate provably useless calls to each existing web service. For example, if we have access to a web service that accepts a *proteinid* of a human protein and provides information about the protein, we should add a filter before calling the web service to ensure that all requests sent to the service are for human proteins. The concept of adding filters before data sources is similar in spirit to 'pushing' selections in the queries in deductive databases [21]. However, the key difference is that the selections 'pushed' by the tuple-level filtering algorithm originate from the source descriptions and not from the user query.

The tuple-level filtering algorithm may also add requests to additional sources as sensing operations to obtain the values of the attributes involved in the constraint. We first convert the datalog program into dataflow-style execution plan using techniques described in Sect. 5. The tuple-level filtering algorithm adds the necessary filters and sensing operations into the dataflow-style execution plan.

Figure 10a shows a graphical representation of a request to a web service (SF) in the parameterized plan. We use a vector notation (capital, boldface) to denote lists of attributes. The web service (SF) accepts a set of inputs ($\mathbf{X_b}$) and provides a set of outputs ($\mathbf{X_b} \bigcup \mathbf{Z}$). The source description of the service (SF) is a conjunction of domain predicates ($\bigwedge_i P_i(\mathbf{X_i})$) and a constraint ($C(Y)$) on attribute Y. In our running example, the web service *HSProtein* is one of the sources being filtered (SF). The only required input to the source is the *proteinid* ($\mathbf{X_b} = [proteinid]$). The source provides *proteinid*, *name*, *location*, *function*, *sequence*, and *pubmedid* attributes ($\mathbf{Z} = [name, location, function, sequence, pubmedid]$). Moreover, there exists a constraint on attribute *taxonid* in the source description ($Y = taxonid$).

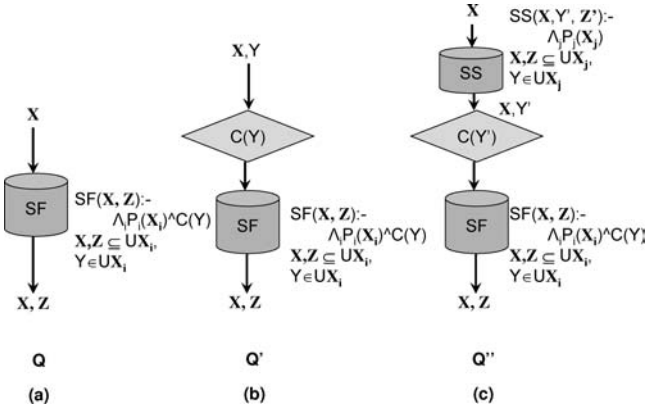

Fig. 10 **a** Initial composition plan, **b** insertion of a filtering operation, and **c** insertion of sensing and filtering operations

Intuitively, we would like to use the constraint $C(Y)$ to insert a *select* operation before the request to the service (SF). As shown in Fig. 10b and c, there are two scenarios: (1) the value of attribute Y is already computed before the request to the service (SF) or (2) the value of attribute Y is not computed before call to the service (SF). In Fig. 10b, the filtering algorithm only needs to add a *select* operation to filter out tuples that do not satisfy constraint $C(Y)$. In Fig. 10c, the filtering algorithm inserts a call to another web service (SS) to obtain value for attribute Y followed by a *select* operation to filter out tuples that do not satisfy constraint $C(Y)$. The tuple-level filtering algorithm accepts an integration plan (similar to Fig. 10a) and if possible inserts sensing and/or filtering operations to obtain a more efficient integration plan (similar to Fig. 10b or c).

Figure 11 shows the tuple-level filtering algorithm. The algorithm first analyzes the generated integration plan to obtain a list of source calls. For each request, the algorithm finds the description of the source. If the description of the source contains a constraint, the algorithm attempts to insert necessary sensing and filtering operations to reduce the number of requests sent to each source. If the value for the attribute involved in the constraint is already present in the plan, the tuple-level filtering algorithm inserts a filtering operation to filter out tuples that conflict with the constraint in the source description. We describe the process of inserting filtering operations (without a sensing operation) in Sect. 4.1.

For some generated plans, the values of the attributes participating in the constraints may not be retrieved before calling the source. In those cases, the tuple-level filtering algorithm may insert sensing services to first obtain the values of those attributes. While this may sound counter-productive at first, it may be helpful since one additional web service request may avoid requests to multiple web services at a later stage in the plan. Section 4.2 describes the process of selecting and adding additional source requests to the generated plan. Section 4.3 proves the correctness of the tuple-level filtering algorithm. Finally, Sect. 4.4 discusses the applicability of our algorithm in the bioinformatics domain.

4.1 Tuple-level filtering without sensing

Intuitively, adding filters to the generated program is a three-step process. First, the algorithm needs to find calls to

Procedure Tuple-level Filtering(SrcDesc, TPlan)
Input: *SrcDesc*: Source Descriptions (LAV rules)
DTPrg: Rules in the Datalog Program
TPlan: Corresponding Theseus plan
Output: Optimized Theseus plan
Algorithm:

1. $SrcPreds$:= headsof$SrcDesc$ /* source predicates */
2. For each call to a source SF in $TPlan$
3. $BoundAttrs$:= attribute values computed by operators before SF in $TPlan$
4. For each constraint C in the source description for SF
5. $Attrs$:= attributes of C
6. If $Attrs \subseteq BoundAttrs$ Then /* insert filtering constraint */
7. insert constraint C before SF in $TPlan$
8. Else **/* insert sensing source predicate */**
9. If ∃ source predicate SS in $SrcPreds$ such that
10. $CompatibleSensingSource(SS, SF,$ TPlan)
11. Then **/* insert sensing operation */**
12. insert predicate SS before SF in $TPlan$
13. insert constraint C before SF in TPlan
14. insert minus operation to find missing tuples
15. due to incompleteness of SS (as shown in Figure 15)
16. union the missing tuples with output of constraint C
17. pass the unioned tuples to SF

Procedure CompatibleSensingSource$(SF, SS, TPlan)$
Input: $SF : SF(\mathbf{X_b}, \mathbf{Z}) :- \bigwedge_i P_i(\mathbf{X_i}) \wedge C(Y)$
where the P_i denote domain predicates,
$\mathbf{X_b}$ are the required input attributes to SF,
$\mathbf{Z} \subseteq \bigcup \mathbf{X_i}$, $\mathbf{X_b} \subseteq \bigcup \mathbf{X_i}$, and $Y \in \bigcup \mathbf{X_i}$.
$SS : SS(\mathbf{X'_Y}, Y', \mathbf{Z'}) :- \bigwedge_j P_j(\mathbf{X_j})$
where the P_j denote domain predicates and
$\mathbf{X'_Y} \subseteq \bigcup \mathbf{X_j}, Y' \in \bigcup \mathbf{X_j}$, and $\mathbf{Z'} \subseteq \bigcup \mathbf{X_j}$.
TPlan: Corresponding Theseus plan
Output: True: if SS is compatible
False: Otherwise
Algorithm:
/* A sensing source SS is *compatible* with a source SF in plan $TPlan$ if */
/* the following conditions are satisfied: */
18. If [$SS \notin TPlan$] and
19. [$\forall X \in \mathbf{X_b}\ \exists X' \in \mathbf{X'_Y}$ such that typeof(X) = typeof(X') (let $\mathbf{X'_Y} = \bigcup X'$)] and
20. [typeof(Y') = typeof(Y)] and
21. [$Q_{SSC} \not\subseteq Q_{SF}$ where
Q_{SSC}: $q(\mathbf{X'_Y}, Y') :- \bigwedge_j P_j(\mathbf{X_j})$
Q_{SF}: $q(\mathbf{X_b}, Y) :- \bigwedge_i P_i(\mathbf{X_i}) \wedge C(Y)$] and
22. [∃ Functional dependencies $\mathbf{X_b} \rightarrow Y$ in $\bigwedge_i P_i(\mathbf{X_i})$ and $\mathbf{X'_Y} \rightarrow Y'$ in $\bigwedge_j P_j(\mathbf{X_j})$] and
23. [$Q_{SFSS} \subseteq Q_{SF}$ where
Q_{SFSS}: $q(\mathbf{X'_Y}, Y') :- \bigwedge_i P_i(\mathbf{X_i}) \wedge C(Y) \bigwedge_j P_j(\mathbf{X_j}) \wedge (\mathbf{X_b} = \mathbf{X'_Y})$]
24. Then Return true
25. Else Return false

Fig. 11 Tuple-level filtering algorithm

all data sources (line 2). For each source call (SF), it first calculates the attributes that are bound to constants, bound to input parameters, or bound to source accesses that have already been performed (line 3). Second, it finds all the attributes involved in the constraints in the source description (line 5). Third, if the values of those attributes are calculated before calling the source, the algorithm inserts the constraint in the integration plan before the source call to filter out tuples that do not satisfy the constraint (line 7). To insert a filter, the algorithm simply adds a *select* operation.

For example, consider a request to create a web service that accepts a *proteinid* and *taxonid* and finds all protein–protein interactions. Figure 12 shows the datalog plan generated by the techniques described in Sect. 3.2. The graphical representation of the parameterized plan generated using the traditional data integration techniques is shown in Fig. 13a. When we use the tuple-level filtering to optimize the generated plan, the filtering algorithm analyzes the generated plan

IR1:ProteinProteinInteractions(fromid, toid, taxonid, source, verified):-
HSProteinInteractions(fromid, toid, source, verified)
IR2:ProteinProteinInteractions(fromid, toid, taxonid, source, verified):-
MMProteinInteractions(fromid, toid, source, verified)
DR:ProteinProteinInteractions(fromid, toid, taxonid, source, verified):-
ProteinProteinInteractions(fromid, itoid, taxonid, source, verified) ∧
ProteinProteinInteractions(itoid, toid, taxonid, source, verified)
Q2:Q(fromid, toid, taxonid, source, verified):-
ProteinProteinInteractions(fromid, toid, taxonid, source, verified) ∧
(fromid = !fromproteinid) ∧
(taxonid = !taxonid)

Fig. 12 Datalog representation of the example composition plan

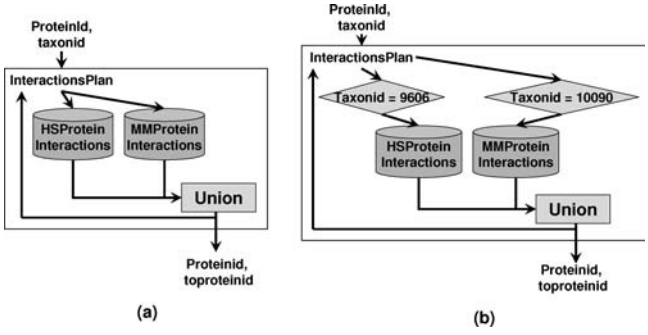

Fig. 13 **a** Initial composition plan and **b** optimized composition plan

and the source descriptions of the *MMProteinInteractions* and *HSProteinInteractions* web service operations. The algorithm uses the constraints on the *taxonid* attribute and adds a filtering constraint before sending requests to each web service operation as shown in Fig. 13b. As the value of the *taxonid* attribute is provided as an input to the composed web service, the filtering algorithm does not need to add any sensing operations.

The value of the *taxonid* attribute is not known at plan generation time. This is the key difference from the traditional query reformulation and optimization techniques that rely on filtering sources by analyzing constraints in the source descriptions and queries. The tuple-level filtering algorithm instead uses filtering operations to encode conditional plans that are similar in spirit to the concept of universal plans [4]. Once the filtering algorithm generates the optimized plan, we utilize a cost-based optimizer to evaluate the cost of the original plan shown in Fig. 13a as well as the optimized plan shown in Fig. 13b. The cost of the plan is calculated by summing the cost of potential requests sent to different services. We define the cost of sending a request to a web service as the response time of the service. The optimizer picks the plan with lower cost (in this case, the optimized plan shown in Fig. 13b) as the composition plan.

4.2 Adding sensing operations

If the values of the attributes participating in the constraints are not retrieved before calling the source, the tuple-level filtering algorithm attempts to insert additional web services to first obtain the values of those attributes. We use the term

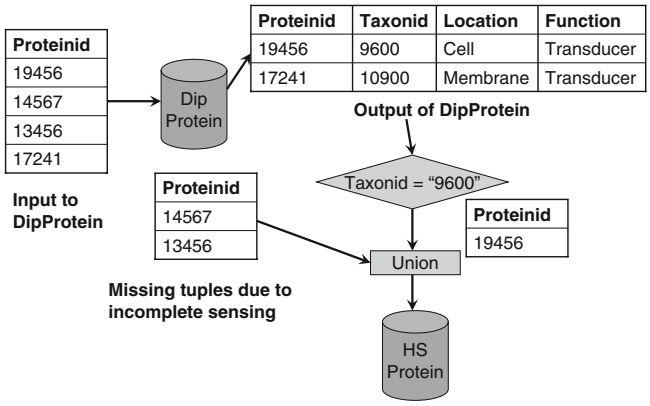

Fig. 14 Example of loss of tuples due to incomplete sensing source

sensing source to refer to such additional web services. The addition of the sensing source can produce a more cost-efficient plan as it can reduce the number of requests sent to the source being filtered. The key criteria for the sensing source are (1) the addition of the sensing service should not change the meaning of the integration plan and (2) the addition of sensing service should lead to a more cost-efficient plan.

As shown in Fig. 10a and c, the modified query (Q'') after the insertion of the sensing and filtering operations is a subset of the original query (Q). Therefore, to ensure that the meaning of the original query does not change, we need to ensure that the insertion of sensing and filtering operations does not lead to removal of qualified tuples. The modified query (Q'') contains two operations that may remove the tuples: (1) the call to the sensing source (SS) and (2) the filtering operation ($C(Y)$).

As we are operating under the open-world assumption, the sensing source may not be complete, i.e. it may not provide a value for attribute Y for all values of the input attributes ($\mathbf{X}_b$). To clarify this point, consider the plan shown in Fig. 14. Imagine that the *DIPProtein* web service only returned values for some input values. Figure 14 shows an example of inputs to the web service and the corresponding outputs. Note that the output of the service is missing some values of the *proteinid* attribute. As some of these missing values may produce qualifying tuples to answer the query, we would like to ensure that those tuples (tuples with values '13456' and '14567') are also passed to the next step. The tuple-level filtering algorithm identifies the missing tuples (lines 14 and 15 of Fig. 11) and unions the missing tuples with the result of the filtering operation (lines 16 and 17 of Fig. 11) to ensure that the sensing operation does not remove any useful tuples.

The tuple-level filtering algorithm also needs to ensure that the filtering operation only removes provably useless tuples. The tuple-level filtering ensures this by requiring that the sensing source satisfies the six conditions shown in the procedure $CompatibleSensingSource$ in Fig. 11. Section 4.2.1 describes the process of selecting compatible sensing sources. Section 4.2.2 describes the process of inserting the selected sensing source(s) and filtering operation(s) in the generated integration plan.

4.2.1 Selecting sources for sensing

When the tuple-level filtering algorithm determines that a sensing operation is needed to obtain the value of an attribute, it uses the *CompatibleSensingSource* procedure to search through available sources (lines 9 and 10 of Fig. 11). All the sources that satisfy all six conditions are returned as available sensing sources. The *CompatibleSensingSource* method finds a sensing source that satisfies six conditions. The first condition (line 18 in Fig. 11) requires that we should not introduce a new sensing operation if it is already present in the plan. The second condition (line 19) requires that the service being used as the sensing source (SS in Fig. 11) must contain attributes (X'_Y) of the same types as the input attributes (X_b) to the source being filtered (SF). Similarly, the third condition (line 20) requires that the sensing source must contain an attribute (Y') of the same type as the attribute that participates in the constraint (Y). Intuitively, if we cannot find attributes of matching types, then the service cannot be used as the sensing source.

The fourth condition (line 21) requires that the description of the service being used as the sensing source (SS) is not a subset of the description of the source being filtered (SF). If the description of the sensing source is a subset of the description of the source being filtered, the insertion of the sensing source and filtering operation would not result in fewer requests to the source being filtered. As we are using the open-world assumption, we do not know if the services are complete. Therefore, we cannot guarantee that the sensing source will definitely remove some tuples. The best we can do is ensure that the sensing operation *may* remove some tuples.

Even if the sensing operation meets the first four conditions, it may not be valid since it may change the meaning of the query. The fifth and the sixth condition in the *CompatibleSensingSource* procedure ensure that the filtering operation ($C(Y')$) has the same meaning as the constraint ($C(Y)$) by assuring that attributes Y and Y' have the same meaning. To clarify the fifth and the sixth conditions, consider the integration plans shown in Fig. 15. The sixth condition in the *CompatibleSensingSource* procedure requires that for all the values of the input attributes ($\mathbf{X}_b$) that satisfy $\bigwedge_i P_i(\mathbf{X}_i) \wedge C(Y)$ (the body of SF) and that satisfy $\bigwedge_j P_j(X_j)$ (the body of SS), the value of Y' is the same as value of Y. This condition is stated as a containment check formula in the sixth condition. The condition checks that all the tuples $[X, Y']$ that satisfy $\bigwedge_i P_i(X_i) \wedge C(Y) \bigwedge_j P_j(X_j)$ (the conjunction of the bodies of SS and SF joined on X), are contained in the set of tuples $[\mathbf{X}, Y]$ that satisfy $\bigwedge_i P_i(\mathbf{X}_i) \wedge C(Y)$. Note that $Y' \in \bigcup \mathbf{X}_j$. The functional dependency requirements in the fifth condition ensure that for any given value of the input attributes to the source being filtered, there is exactly one value for the attributes involved

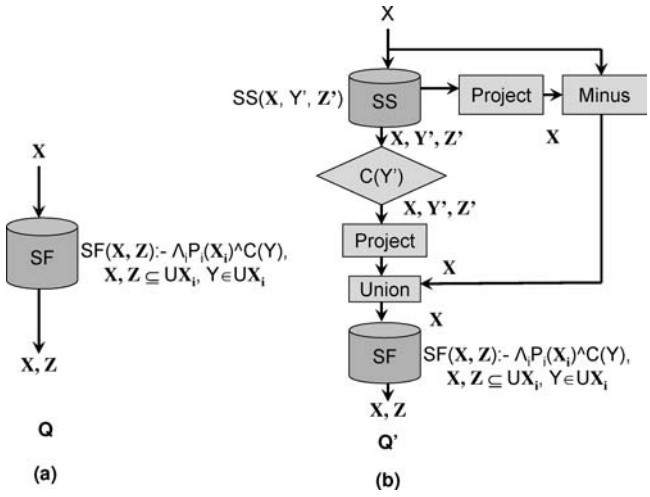

Fig. 15 Example partial integration plans **a** before insertion of sensing and filtering operations and **b** after insertion of sensing and filtering operations

in the constraint. So, given the functional dependencies, the sixth condition is only satisfied when attributes Y (in SF) and Y' (in SS) have the same meaning.

As an example, consider the datalog rules shown in Fig. 6 and the query rule shown in Fig. 8. The graphical representation for the datalog program is shown in Fig. 9. The tuple-level filtering algorithm begins the optimization by analyzing the generated plan. There are 10 source calls in the generated plan: two instances of *HSProtein*, *MMProtein*, *MembraneProtein*, and *TransducerProtein* and one instance of *HSProteinInteractions* and *MMProteinInteractions*. The source *HSProtein*, which contains equality constraint on the attribute *taxonid*. However, the *taxonid* attribute is not one of the attributes retrieved before the call to the source. At this point, the optimization algorithm searches through the list of sources to find a sensing source compatible with *HSProtein* (lines 9–12 of Fig. 11).

In the given example, the algorithm finds the source *DIPProtein* that is not in the integration plan (satisfying first condition from Fig. 11). The *DIPProtein* source accepts a *Proteinid* and provides a *taxonid* (this satisfies the second and third conditions). The *DIPProtein* source also satisfies the fourth condition as the *Protein* domain relation contains all the proteins ($\bigwedge_i P_j(\mathbf{X}_j) = Protein(\cdot)$). Also, the *proteinid* functionally determines *taxonid* (which satisfies the fifth condition).

For the *DIPProtein* and *HSProtein* sources, the sixth condition is

$$
\begin{aligned}
&Q_{SFSS} \subseteq Q_{SF} \text{ where}\\
&Q_{SFSS} : q(id, taxonid') : -\\
&\qquad /* \textstyle\bigwedge_i P_i(\mathbf{X}_i) */\\
&\qquad Protein(id, name, location, function,\\
&\qquad\qquad sequence, pubmedid, taxonid) \wedge\\
&\qquad /* C(Y) */\\
&\qquad taxonid = 9606 \wedge\\
&\qquad /* \textstyle\bigwedge_j P_j(\mathbf{X}_j) */\\
&\qquad Protein(id', name', location', function',\\
&\qquad\qquad sequence', pubmedid', taxonid') \wedge\\
&\qquad /* \mathbf{X}_b = \mathbf{X}'_Y */\\
&\qquad id = id'\\
&Q_{SF} : q(id, taxonid) : -\\
&\qquad /* \textstyle\bigwedge_i P_i(\mathbf{X}_i) */\\
&\qquad Protein(id, name, location, function,\\
&\qquad\qquad sequence, pubmedid, taxonid) \wedge\\
&\qquad /* C(Y) */\\
&\qquad taxonid = 9606
\end{aligned}
$$

We use the methods described in [22] to determine that Q_{SFSS} is contained in Q_{SF} given the functional dependencies. Intuitively, given that the *DIPProtein* data source satisfies the functional dependency requirements, the id attribute in the *Protein* domain relation functionally determines the *taxonid* attribute. Similarly, the id' attribute functionally determines the value of the $taxonid'$ attribute. Given that the id and id' attributes have the same value in Q_{SFSS}, *taxonid* and $taxonid'$ attributes also have the same value. Therefore, we can rewrite Q_{SFSS} by unifying the two instances of the *protein* relation as shown before.

$$
\begin{aligned}
&Q_{SFSS} \subseteq Q_{SF} \text{ where}\\
&Q_{SFSS} : q(id, taxonid') : -\\
&\qquad Protein(id, name, location, function,\\
&\qquad\qquad sequence, pubmedid, taxonid') \wedge\\
&\qquad taxonid' = 9606\\
&Q_{SF} : q(id, taxonid) : -\\
&\qquad Protein(id, name, location, function,\\
&\qquad\qquad sequence, pubmedid, taxonid) \wedge\\
&\qquad taxonid = 9606
\end{aligned}
$$

Once we rewrite the query Q_{SFSS}, it is clear that Q_{SFSS} is contained in Q_{SF}. Therefore, the *DIPProtein* data source satisfies the sixth condition.

Since the *DIPProtein* data source matches all the conditions in the procedure *CompatibleSensingSource*, the filtering algorithm selects the *DIPProtein* data source as a sensing operation.

The filtering algorithm does not use the *ProteinLocations* data source as it requires the *name* of the protein in addition to the *proteinid* and the value for the *name* attribute has not been retrieved.

Consider an example service called *ClosestOrthologSrc* that satisfies the first five conditions of the tuple-level filtering algorithm, but not the critical sixth condition. The *ClosestOrthologSrc* service accepts a *Proteinid* and returns the *taxonid* for the organism with the closest ortholog to the protein. The *taxonid* returned by the *ClosestOrthologSrc* is the *taxonid* of a different protein. Therefore, the tuple-level filtering should not use the *ClosestOrthologSrc* as a sensing operation before *HSProtein* service. We can describe this source using the following source description:

ClosestOrthologSrc(id, otaxonid):-
Protein(id, name, location, function, sequence, pubmedid, taxonid) ∧
Protein(oid, oname, oloc, ofunction, osequence, opubmedid, otaxonid) ∧
ClosestOrthologProtein(id, oid)

The domain predicate *ClosestOrthologProtein* contains information about the closest ortholog protein for each protein. As there is only one closest ortholog protein for each protein, the attribute *id* functionally determines the attribute *oid*. Moreover, for the source *ClosestOrthologSrc* the attribute *id* functionally determines the attribute *otaxonid*. Given this scenario, it seems like tuple-level filtering may select the *ClosestOrthologSrc* service as a sensing source before the *HSProtein* service.

The *ClosestOrthologSrc* service is not in the plan, so it satisfies the first condition. The *id* attribute in the *ClosestOrthologSrc* service has the same type as the *proteinid* attribute in the *HSProtein* service and the *otaxonid* attribute in the *ClosestOrthologSrc* service has the same type as the *taxonid* attribute. Therefore, the *ClosestOrthologSrc* service satisfies the second and third conditions. Also, the description of the *ClosestOrthologSrc* service does not have a conflicting constraint on the attribute *otaxonid*. Therefore, the *ClosestOrthologSrc* service satisfies the fourth condition. There exists a functional dependency between the *id* attribute and the *otaxonid* attribute, which satisfies the functional dependency requirement in the fifth condition.

However, the *ClosestOrthologSrc* data source does not satisfy the sixth condition. Recall that the sixth condition states that:

$Q_{SFSS} \subseteq Q_{SF}$ where
$Q_{SFSS} : q(\mathbf{X}, Y') :- \bigwedge_i P_i(\mathbf{X}_i) \wedge C(Y) \bigwedge_j P_j(\mathbf{X}_j) \wedge (\mathbf{X}_b = \mathbf{X}'_Y)$
$Q_{SF} : q(\mathbf{X}, Y) :- \bigwedge_i P_i(\mathbf{X}_i) \wedge C(Y)$

Replacing the values from the descriptions of services,

$Q_{SFSS} \subseteq Q_{SF}$ where
$Q_{SFSS} : q(id, otaxonid) :-$
$/* \bigwedge_i P_i(\mathbf{X}_i) */$
Protein($id, name, location, function, sequence, pubmedid, taxonid$)∧
$/* C(Y) */$
$taxonid = 9606 \wedge$
$/* \bigwedge_j P_j(\mathbf{X}_j) */$
$Protein(id1, name1, location1, function1, sequence1, pubmedid1, taxonid1) \wedge$
$Protein(oid, oname, oloc, ofunction, osequence, opubmedid, otaxonid) \wedge$
$ClosestOrthologProtein(id, oid) \wedge$
$/*(\mathbf{X}_b = \mathbf{X}'_Y)*/$
$id = id1$

$Q_{SF} : q(id, taxonid) :-$
$/* \bigwedge_i P_i(\mathbf{X}_i) */$
Protein($id, name, location, function, sequence, pubmedid, taxonid$)∧
$/* C(Y) */$
$taxonid = 9606$

However, our system can prove using techniques described in [22] to prove that Q_{SFSS} is not contained in Q_{SF}. Therefore, the tuple-level filtering algorithm does not select the *ClosestOrtholog* service as a sensing source.

4.2.2 Inserting sensing and filtering operations in the plan

Once the tuple-level filtering determines the compatible sensing source(s), it inserts a request(s) to each qualifying sensing source followed by a filter (lines 13 and 14 in Fig. 11) before the request to the source being filtered. If there are multiple compatible sensing sources, the tuple-level filtering algorithm inserts requests to all of the sensing sources followed by a filter before the request to the source being filtered. In our running example, the tuple-level filtering algorithm inserts a request to the *DIPProtein* data source followed by a constraint $taxonid' = 9606$ before the request to the *HSProtein* data source. Similar filters are also introduced before sending requests to *MMProtein*, *MembraneProtein*, and *TransducerProtein* sources.

The optimized program for the running example is shown in Fig. 16. For clarity, we have shown the filters and the retrieval operations for different protein sources separately in Fig. 17.[4] The optimized plan first sends request to the *DIPProtein* source to obtain the *taxonid, location*, and *function* information. Then, filters based on *taxonid, location*, and *function* attributes are used to determine which protein sources should be queried to obtain the protein information for the given *proteinid*. Filters based on the *taxonid* attribute are also used to determine which protein–protein interactions source should be queried. For all the interacting proteins, a similar process is repeated.

In this example, the algorithm only needs to add one sensing operation for all sources as all the necessary attributes can be obtained from the *DIPProtein* data source. However, in some scenarios the algorithm may need to add multiple sensing operations. Once the filtering algorithm generates the optimized plan, we utilize a cost-based optimizer to evaluate the cost of the original plan as well as the optimized plan. The optimizer picks the plan with less cost (in this case, the optimized plan shown in Fig. 16) as the composition plan.

4.3 Correctness of tuple-level filtering

In this section, we show that the sensing operations inserted by the tuple-level filtering algorithm do not change the answer of the query.

[4] As a matter of fact, our execution architecture, Theseus, allows for the encapsulation of sets of operations into reusable subplans.

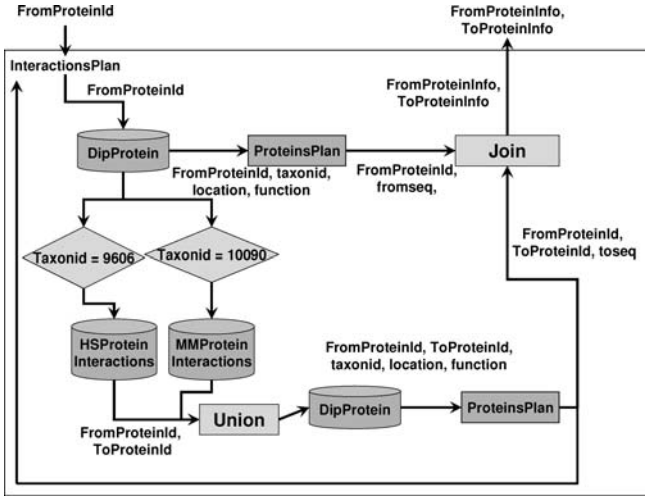

Fig. 16 Optimized integration plan

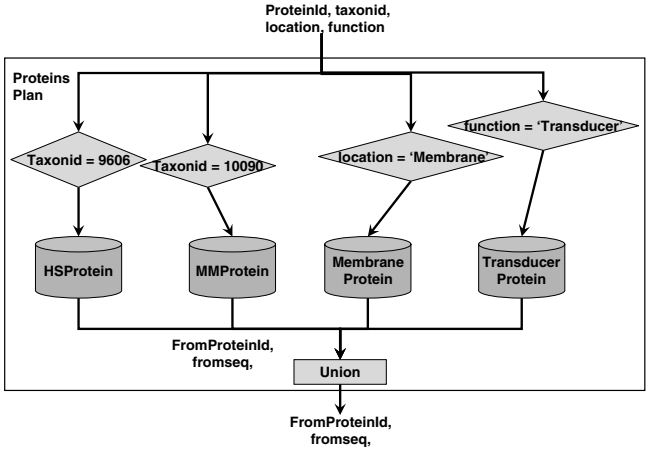

Fig. 17 Proteins plan called from the integration plan in Fig. 16

Theorem 1 *Given an integration plan Q generated using the Inverse Rules algorithm to answer the user query, the tuple-level filtering algorithm produces a new integration plan Q' containing sensing operations and filters such that $Q \equiv Q'$.*

Proof Consider the partial integration plan before and after adding sensing operation shown in Fig. 15. The data source SF is part of an integration plan generated to answer a user query. The source SF is described as a conjunction of domain predicates $P_i(\mathbf{X}_i)$ and a constraint $C(Y)$. Without loss of generality, we assume that the source description only contains one equality constraint. Assume that the tuple-level filtering algorithm inserted before SF a sensing source SS and a selection that enforces $C(Y)$. Thus, SS satisfies the conditions in Fig. 11. Recall the definitions of sources SF and SS:

$$SF(\mathbf{X}_b, \mathbf{Z}) :- \bigwedge_i P_i(\mathbf{X}_i) \wedge C(Y)$$
$$SS(\mathbf{X}'_Y, Y', \mathbf{Z}') :- \bigwedge_j P_j(\mathbf{X}_j)$$

Query Q later shows the plan before the insertion of the sensing operation. The relation $R(\mathbf{X})$ represents the inputs into SF from the preceding operations of the plan. Query Q' represents the plan after the insertion of the sensing operation SS. The first rule of Q' corresponds to the case when SS contains a tuple for the given value of $\mathbf{X}$, while the second rule represents the case where SS does not contain a tuple for the given value of $\mathbf{X}$. Note that $\mathbf{X} = \mathbf{X}_b = \mathbf{X}'_Y$.

$$Q : q(\mathbf{X}, \mathbf{Z}) :- R(\mathbf{X}) \wedge SF(\mathbf{X}, \mathbf{Z})$$
$$Q' : q(\mathbf{X}, \mathbf{Z}) :- R(\mathbf{X}) \wedge SS(\mathbf{X}, Y', \mathbf{Z}') \wedge C(Y') \wedge SF(\mathbf{X}, \mathbf{Z})$$
$$q(\mathbf{X}, \mathbf{Z}) :- R(\mathbf{X}) \wedge \neg SS(\mathbf{X}, Y', \mathbf{Z}') \wedge SF(\mathbf{X}, \mathbf{Z})$$

First, we show that $Q' \subseteq Q$. Assume that tuple $[\mathbf{X}, \mathbf{Z}] \in$ Q'; the tuple $[\mathbf{X}, \mathbf{Z}]$ is produced by either the first or the second rule of Q'. We analyze both cases:

1. Assume the tuple is an output of the first rule of Q', that is, $[\mathbf{X}, \mathbf{Z}] \in R(\mathbf{X}) \wedge SS(\mathbf{X}, Y', \mathbf{Z}') \wedge C(Y') \wedge SF(\mathbf{X}, \mathbf{Z})$. Since the tuple satisfies the entire conjunctive formula, it also satisfies the subformula: $[\mathbf{X}, \mathbf{Z}] \in R(\mathbf{X}) \wedge SF(\mathbf{X}, \mathbf{Z})$. Since this is the body of Q, then $[\mathbf{X}, \mathbf{Z}] \in Q$.
2. Assume the tuple is an output of the second rule of Q', that is, $[\mathbf{X}, \mathbf{Z}] \in R(\mathbf{X}) \wedge \neg SS(\mathbf{X}, Y', \mathbf{z}') \wedge SF(\mathbf{X}, \mathbf{z})$. As before, since the tuple satisfies the entire conjunctive formula, it also satisfies the subformula $[\mathbf{X}, \mathbf{z}] \in R(\mathbf{X}) \wedge SF(\mathbf{X}, \mathbf{z})$, which is the body of Q. Thus, $[\mathbf{X}, \mathbf{z}] \in Q$.

Therefore, $Q' \subseteq Q$. The insertion of the sensing operation and the filter by the algorithm does not introduce additional tuples.

Second, we show that $Q \subseteq Q'$. Assume that tuple $[\mathbf{X}, \mathbf{z}] \in Q$. Then, by the definition of Q,

$$[\mathbf{X}, \mathbf{z}] \in R(\mathbf{X}) \wedge SF(\mathbf{X}, \mathbf{z}) \tag{1}$$

Given the functional dependencies $\mathbf{X} \rightarrow Y$ and $\mathbf{X} \rightarrow Y'$ and the definition of Q', we need to consider three cases: either a tuple in Q is not in SS, or it is in SS and satisfies $C(Y')$, or it is in SS and does not satisfy $C(Y')$.

1. Assume that $\exists Y', \mathbf{z}'$ such that $[\mathbf{X}, Y', \mathbf{z}'] \notin SS$.
   Then, from (1) and the assumption in this case, the tuple $[\mathbf{X}, \mathbf{z}]$ satisfies the body of the second rule for Q', that is, $[\mathbf{X}, \mathbf{z}] \in R(\mathbf{X}) \wedge SF(\mathbf{X}, \mathbf{z}) \wedge \neg SS(\mathbf{X}, Y', \mathbf{z}')$. Therefore, $[\mathbf{X}, \mathbf{z}] \in Q'$.
2. Assume that $\exists Y', \mathbf{z}'$ such that $[\mathbf{X}, Y', \mathbf{z}'] \in SS \wedge C(Y')$.
   Then, from (1) and the assumption in this case, the tuple $[\mathbf{X}, \mathbf{z}]$ satisfies the body of the first rule for Q', that is, $[\mathbf{X}, \mathbf{z}] \in R(\mathbf{X}) \wedge SF(\mathbf{X}, \mathbf{z}) \wedge SS(\mathbf{X}, Y', \mathbf{z}') \wedge C(Y')$. Therefore, $[\mathbf{X}, \mathbf{z}] \in Q'$.
3. Assume that $\exists Y', \mathbf{z}'$ such that $[\mathbf{X}, Y', \mathbf{z}'] \in SS \wedge \neg C(Y')$.
   Expanding the definition of SS,

$$[\mathbf{X}, Y', \mathbf{z}'] \in \bigwedge_j P_j(\mathbf{X}_j) \wedge \neg C(Y') \tag{2}$$

   By assumption, tuple $[\mathbf{X}, \mathbf{z}] \in Q$. Therefore, tuple $[\mathbf{X}, \mathbf{z}]$ satisfies (1). Thus, it also satisfies the definition of SF:

$$\exists Y \ [\mathbf{X}, Y, \mathbf{z}] \in \bigwedge_i P_i(\mathbf{X}_i) \wedge C(Y) \tag{3}$$

From Eqs. (2) and (3), we have that:

$$[\mathbf{X}, Y'] \in \bigwedge_j P_j(\mathbf{X}_j) \wedge \neg C(Y') \bigwedge_i P_i(\mathbf{X}_i) \wedge C(Y) \quad (4)$$

(Note that the formula joins on $\mathbf{X}$. Recall that $\mathbf{X} = \mathbf{X}_Y = \mathbf{X}_b$, $\mathbf{X}_b \subseteq \bigcup \mathbf{X}_i$, $\mathbf{X}'_Y \subseteq \bigcup \mathbf{X}_j$, $Y' \in \bigcup \mathbf{X}_j$ and $Y \in \bigcup \mathbf{X}_i$.) As SS was chosen as the sensing source by tuple-level filtering, it must satisfy condition 6 in procedure *CompatibleSensingSource* in Fig. 11:

$Q_{SFSS} \subseteq Q_{SF}$ where
Q_{SFSS}: $q(\mathbf{X}, Y')$:- $\bigwedge_i P_i(\mathbf{X}_i) \wedge C(Y) \bigwedge_j P_j(\mathbf{X}_j)$
Q_{SF}: $q(\mathbf{X}, Y)$: - $\bigwedge_i P_i(\mathbf{X}_i) \wedge C(Y)$

By definition of Q_{SF} $\forall \mathbf{X}, Y [\mathbf{X}, Y] \in Q_{SF}$, Y satisfies $C(Y)$. However, from (4), there exists a tuple $[\mathbf{X}, Y']$ such that $[\mathbf{X}, Y'] \in Q_{SFSS}$ and Y' satisfies $\neg C(Y')$. Therefore, there exists a tuple $[\mathbf{X}, Y'] \in Q_{SFSS}$ that is not present in Q_{SF}. Thus, $Q_{SFSS} \not\subseteq Q_{SF}$, which is a contradiction.

Therefore, $Q \subseteq Q'$.
Since, $Q' \subseteq Q$ and $Q \subseteq Q'$, then $Q \equiv Q'$. □

4.4 Tuple-level filtering in the bioinformatics domain

In this section, we discuss the applicability of the tuple-level filtering in the bioinformatics domain. In particular, we show examples of real-world data sources and domain models where the tuple-level filtering results in more cost efficient plans. As discussed in Sect. 4.2, tuple-level filtering requires that the sensing source must meet six conditions. The fifth and sixth conditions of the tuple-level filtering are the key conditions that guarantee the correctness of the optimized plan.

The fifth condition states that the input attributes ($\mathbf{X}$) to the source being filtered must functionally determine the attribute involved in the constraint (Y). Moreover, the same relationship should hold between the corresponding attributes in the sensing source (SS) and the attribute (Y') used in the constraint for the filtering operation. In the life sciences domain, most data sources provide at least some attribute(s) that serves as a local key that identifies different entities. The attribute that serves as the local key often functionally determines other attributes. The existence of the functional dependency implies that the fifth condition of the tuple-level filtering would be satisfied for a large number of bioinformatics sources.

The sixth condition requires that the sensing source provides information about the same type of entity as the source being filtered. In the bioinformatics domain, there exists a variety of data sources that provide detailed information about different entities and have a well-defined coverage. For example, the Human Protein Reference Database (HPRD)[5] provides detailed information about human proteins. Moreover, there exists a set of sources for different entity types that have very good coverage. For example, the UniProt[6] data source provides information about proteins in different organisms. However, UniProt does not provide information about the interactions between different proteins. If the user query was to find out information about all the proteins that the given protein interacts with, the UniProt data source would not be useful to answer the user query. However, UniProt may be a good sensing source to filter out tuples before sending requests to the HPRD data source, as both sources provide protein information. The existence of the sources that provide information about the same type of entities, but have different coverage implies that the sixth condition of the tuple-level filtering would be satisfied by a large number of sources.

Table 2 Available web services

Source
UniProt($accession, creationdate, proteinname, genename, organism, taxonomy, sequence, checksum)
PathCalling($accession, interactingproteinid, typeofinteraction)
HPRD($accession, interactingproteinid, publicationid)

Table 3 Domain predicates

Domain relations
Protein(accession, creationdate, proteinname, genename, organism, taxonomy, sequence, checksum)
Protein-ProteinInteractions(proteinid, interactingproteinid, typeofinteraction, publication)

Consider the three real-world datasets shown in Table 2. The UniProt dataset contains detailed information about different proteins. The PathCalling[7] dataset contains information about the interactions between yeast proteins, while the HPRD dataset contains information about interactions between human proteins.

Our domain model contains the two domain predicates shown in Table 3. Figure 18 shows the source descriptions. Notice that the descriptions of the PathCalling and the HPRD sources include a constraint on the organism.

Given these domain relations, sources, and source descriptions, the user specifies the following parameterized query.

$Q1$(proteinid, interactingproteinid):
ProteinProteinInteractions(proteinid, interactingproteinid, typeofinteraction, publication)∧
proteinid = !proteinid

Given this query, the initial plan generated by the integration system only contains requests to the HPRD and PathCalling data sources. However, after applying tuple-level filtering, the optimized plan first obtains the organism information from the UniProt data source and uses that to filter

[5] http://www.hprd.org/.

[6] http://www.pir.uniprot.org/.

[7] http://curatools.curagen.com/cgi-bin/com.curagen.portal.servlet.PortalYeastList.

```
SD1: UniProt(accession, creationdate, proteinname, genename, organism,
         taxonomy, sequence, checksum):-
     Protein(accession, creationdate, proteinname, genename, organism,
         taxonomy, sequence, checksum)
SD2: PathCalling(proteinid, interactingproteinid, typeofinteraction):-
     Protein(proteinid, creationdate, proteinname, genename, organism,
         taxonomy, sequence, checksum)∧
     Protein(interactingproteinid, icreationdate, iproteinname, igenename,
         iorganism, itaxonomy, isequence, ichecksum)∧
     Protein-ProteinInteractions(proteinid, interactingproteinid,
         typeofinteraction, publication)∧
     organism = 'Saccharomyces cerevisiae (Baker's yeast)'
SD3: HPRD(proteinid, interactingproteinid, publication):-
     Protein(proteinid, creationdate, proteinname, genename, organism,
         taxonomy, sequence, checksum)∧
     Protein(interactingproteinid, icreationdate, iproteinname, igenename,
         iorganism, itaxonomy, isequence, ichecksum)∧
     Protein-ProteinInteractions(proteinid, interactingproteinid,
         typeofinteraction, publication)∧
     organism = Homo Sapiens
```

Fig. 18 Source descriptions

out tuples before sending requests to the HPRD or the PathCalling data sources.

In the bioinformatics domain, there exists a variety of sources that provide information about the same entities, but have different coverage. The TIGRFAM[8] data source organizes the protein information by the function of proteins. In addition to protein information sources, a similar set of sources exists for gene mutation information. Moreover, all of these sources provide some form of local key that functionally determines the other attributes.

Another challenge in bioinformatics domain is to uniquely identify various entities. In particular, when integrating data from various data sources one needs to have a mapping between local keys of different sources to accurately identify entities. For example, when combining data from UniProt[9] and NCBI Protein, we would need to obtain the accession number in the NCBI Protein Database for each protein in UniProt. While several sources provide links to other datasets, those links are often not complete. Nevertheless, the tuple-level filtering can handle incomplete sensing sources. As long as there are several sources that share the local key attributes, the tuple-level filtering algorithm would result in more cost-efficient plans.

As the bioinformatics domain is an active area of research, information about entities changes frequently. For example, gene symbols are often retired and replaced with new symbols (often called aliases). When integrating information from various datasets, one would need to worry about different aliases and synonyms. While the problem of managing identity of objects is very different from the problem of generating efficient composition plans, it may impact the effectiveness of the tuple-level filtering. We can handle this problem by managing the mappings between the local key attributes of different sources in similar spirit to the work described in [23]. We believe that our integration system is well suited for such extension. In particular, we have done some work on automatically utilizing additional sources to accurately link records from different sources [24].

[8] http://www.tigr.org/TIGRFAMs/.

[9] http://www.pir.uniprot.org/.

5 Efficient execution of composition plans

The generated integration plans may send several requests to the existing web services. We can reduce the execution time of the generated plans by executing the generated plans using a streaming, dataflow-style execution engine. The dataflow-style execution engines stream data between operations and execute multiple operations in parallel (if the operations are independent). There has been some work on mapping datalog integration plans into plans that can be executed by dataflow-style execution engines [25]. However, the mapping described in [25] is restricted to non-recursive datalog programs.

We address this limitation by describing our techniques to map recursive and non-recursive integration plans into a dataflow-style execution engine called Theseus [26]. We selected the Theseus execution engine [26] for its two unique features: (1) its declarative plan language and (2) its support for recursive composition plans. First, we will briefly introduce the plan language utilized by the Theseus execution engine. Next, we will describe the translation of non-recursive datalog programs to the Theseus plans. Finally, we will describe the translation of recursive datalog programs.

5.1 Brief introduction to Theseus

A Theseus plan consists of a graph of operations that accepts a set of input relations and produces a set of output relations. A relation in Theseus is similar to relations in relational databases, consisting of a list of attributes and a set of tuples. Theseus streams tuples of the relations between various operations to reduce the runtime of the plan.

Theseus supports a wide variety of operations. The operations relevant to this article can be divided in three sets: (1) operators that support relational manipulations, such as *select*, *project*, *union*, or *join*, (2) data access operations, such as *dbquery* or *retrieve* operation, to retrieve data from databases, wrappers or web services, and (3) conditional operations, such as *null*, to determine the next action based on the existence of data in some relation. All Theseus operations accept one or more input relations, some arguments if needed, and produce an output relation. For example, a *select* operation accepts an input relation and a selection condition and produces an output relation with the tuples that satisfy the criteria.

Another key feature of the plan language of Theseus is the ability to call another Theseus plan from inside a plan. Moreover, Theseus allows the user to write plans that call themselves recursively. As we will show in Sect. 5.3, this allows us to translate recursive datalog programs into plans that can be executed by the Theseus execution engine.

5.2 Mapping composition plans into dataflow programs

If the composed datalog program does not have recursive rules, the translation is relatively straightforward. The translation begins by macro-expanding the datalog rule for the parameterized query, until all the predicates in the rule(s) are data sources or constraints. The mediator then utilizes the translations described in the rest of this section to translate the expanded rule into a Theseus plan. Figure 19 shows examples of different datalog operations and corresponding Theseus plans. The translated Theseus plans in its plan language are shown in Appendix 1.

5.2.1 Data access

Data access predicates to obtain data by sending a request to a web service are translated to retrieval operations in a Theseus plan. For example, *DIPProtein*(‘*19456*’, *name*, *function*, *location*, *taxonid*) denotes a request to the *DIPProtein* web service.

Figure 19a shows an example translation of a data access predicate to a retrieval operation. A retrieval operation in Theseus accepts an optional input relation containing values of necessary inputs for the web service, submits a request to the web service, obtains the result, and returns the resulting information in the form of a output relation. A data access predicate may include constants in the attribute list for a relation. A data access predicate containing with a constant value for an attribute having a binding constraint, is translated to a retrieval operation with the constant as the input parameter value. For example, DIPProtein(‘19456’, name, function, location, taxonid) is translated to a retrieval call with inputs proteinid = ‘19456’ (operation 1 in Fig. 19a). If the attribute list of the relation in the data access predicate contains a constant for a free attribute, then the data access statement is translated to a retrieval operation followed by a select operation as shown in Fig. 19b.

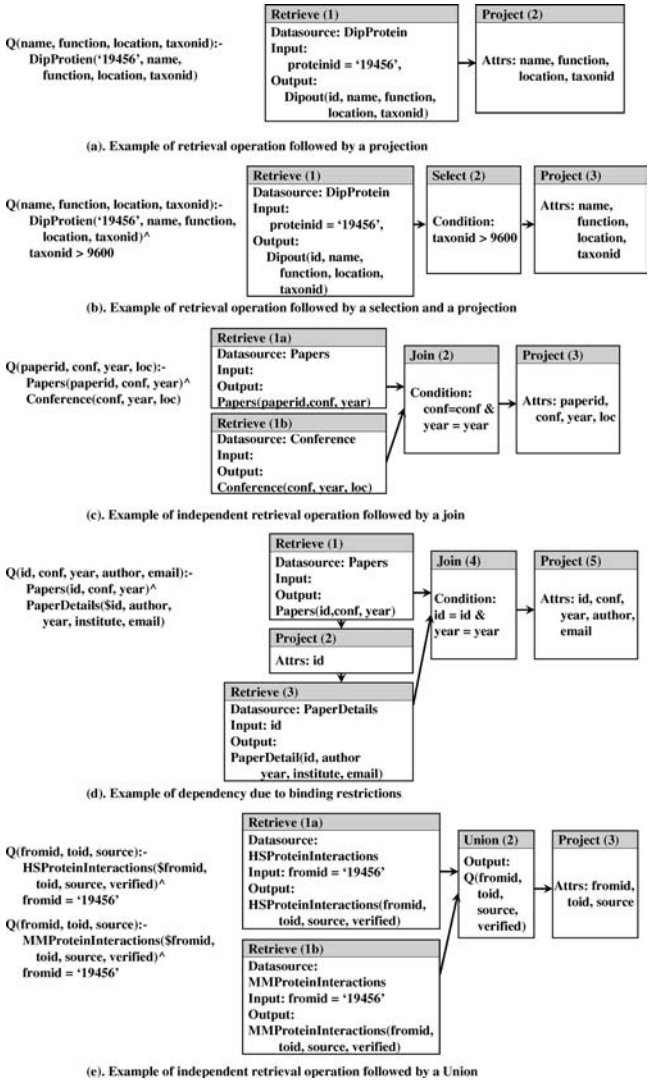

Fig. 19 Example mapping between datalog and Theseus

5.2.2 Select

Equality and order constraints, such as $(x = 5)$ or $(x > y)$ are translated into a select operations. The select operation accepts a relation and a select condition and provides a new relation that contains tuples that satisfy the selection condition. In the example given in Fig. 19b, the select predicate (taxonid > 9600) is translated to a select operation (operation 2).

5.2.3 Project

A project operation in datalog is denoted by variables in the head of a rule. The project operation in data translates to a project operation in Theseus. The project operation in Theseus accepts a relation and attributes to be projected and provides a new relation consisting of tuples with the specified attributes. In the example given in Fig. 19a, Q(name, function, location, taxonid) is translated to a project operation (operation 2). The arrow between operations 1 and 2 denotes the dataflow, i.e. the output of the operation 1 is provided is input to the operation 2. Intuitively, we cannot perform the project operation until we have obtained at least one tuple from the retrieval operation. Once the retrieval operation returns the first tuple, it can be streamed to the project operation. Similar to the select operation, the project operation also depends on the retrieval operation.

5.2.4 Join

A datalog statement containing two relations with one or more common attribute names specifies a join. If the common attribute name in the join is a free attribute in both relations, then the join is replaced by a join operation in the Theseus plan. A join operation in Theseus accepts two relations and a join condition, and outputs a joined relation. Figure 19(c) shows an example of translating a join between

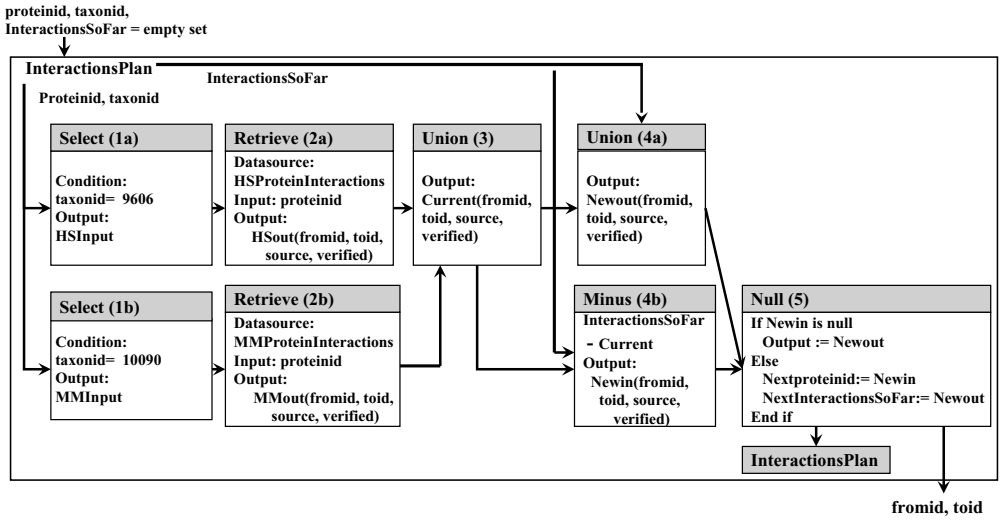

Fig. 20 Example recursive Theseus plan

two data sources resulting in a two independent retrieval operations followed by a join operation.

If the common attribute in the join has a binding constraint in one of the relations, then the join is translated into a dependency between two operations in Theseus. In the example shown in Fig. 19d, there is a join between the *Paper* and the *PaperDetails* predicates on the attributes *id* and *year*. The *id* attribute is a required input for the *PaperDetails* data source. Therefore, the generated Theseus plan first obtains *id*, *conf* and *year* for all papers, projects the *id* attribute, and utilizes the values of the *id* attribute to obtain information from *PaperDetails* data source. These operations are followed up by a join operation on *id* and *year* attributes.

5.2.5 Union

In datalog, two rules having the same head represent a union. The union in datalog is translated to a union operation in Theseus. The union operation in Theseus operation accepts two or more relations as input and provides one output relation that is the union of the given relations. Figure 19e shows an example of a translation of a union operation.

5.3 Translating recursive plans to dataflow programs

The recursive datalog rules are translated to recursive Theseus plans. The recursive Theseus plans are typically divided in five parts: (1) data processing, (2) result accumulation, (3) loop detection, (4) termination check, and (5) recursive callback. Figure 20 shows an example recursive plan obtained by generating the optimized Theseus plan for the example shown in Fig. 13b (corresponding to the datalog rules of Fig. 12). The same plan is shown in the Theseus' plan language in Appendix 1.6.

The first part of a recursive Theseus plan is data processing. Data processing in a recursive Theseus plan may involve accessing data from a data source and processing the data. In the example Theseus plan shown in Fig. 20, operations 1a, 1b, 2a, 2b, and 3 perform data processing. This part typically corresponds to the non-recursive part of the datalog statement and is translated in the same manner as the non-recursive datalog statements. In this case, the two filters based on the *taxonid* attribute are translated to *select* operations 1a and 1b. The data access operations to retrieve data from the *HSProteinInteractions* and *MMProteinInteractions* are translated to the *retrieval* operations 2a and 2b. A *union* operation is used to combine the information from both data sources.

The second part of the recursive Theseus plan is the update of the cumulative results. A recursive Theseus plan needs to keep track of all results that have been acquired through recursion. In our example, this part is responsible for adding all tuples from the *current* and *InteractionsSoFar* relations to the relation *Newout* using a union operation as shown in the operation 4a of the example plan in Fig. 20.

The third part of the recursive plan is loop detection. In datalog, the interpreter is responsible for handling loops. Therefore, the datalog programs do not require explicit statements to perform loop detection. Theseus does not automatically handle loop detection. Therefore, when translating datalog programs to Theseus plans, the mediator must add Theseus operations to handle loop detection. Intuitively, recursion can be viewed as a graph traversal problem, where each recursive step is to follow an edge from one node in the graph to the other. We handle loop detection in recursive plans by keeping track of all visited nodes in the graph and in each recursive step only follow the edges that lead to unvisited nodes.

Each tuple in the output relation of the Theseus plan defines a node in the graph. The tuples in the output of the previous iteration are all the visited nodes. In the example

Theseus plan shown in Fig. 20, *InteractionsSoFar* is the relation containing tuples collected during previous iterations. We can obtain the list of unvisited nodes by removing all the visited nodes, i.e. by removing tuples existing in the *InteractionsSoFar* relation, from the list of tuples obtained in the data processing segment (i.e. the tuples obtained by the union operation shown in the operation 3 of the Fig. 20). A minus operation in Theseus is used to filter out all tuples in one relation from the other relation. A minus operation in Theseus accepts two relations and returns a new relation that contains only the tuple that are present in the first relation. In the given example, a minus operation (4b) is used to filter out tuples with previously seen protein interactions.

Once the plan has determined the new tuples for the next iteration (this may be an empty set), Theseus needs to check for the termination condition to determine if the plan should be recursively called again or the plan should terminate and provide the cumulative output as output. When translating datalog programs to Theseus plans, the termination condition is satisfied when no new input tuples can be found for the next iteration. In the example plan, the *null* operation (operation 5 of Fig. 20) checks for the termination condition. The *null* operation in Theseus accepts three relations, if the first relation is contains no tuples, the second relation is returned, otherwise the third relation is returned. In this case, if there are no new tuples, the *null* operation returns all the tuples collected as cumulative results as the output of the recursive plan. If there are new tuples, the *null* operation recursively calls the plan with the new proteins as well as the cumulative results found in the second part.

5.4 Execution of the composed web services

Once the mediator has translated the datalog program for the composed web service into a Theseus plan, it can host the generated plan as a web service. When a user sends a request to the composed web service, the Theseus execution engine executes the Theseus plan using the parameters given by the user. In this section, we describe the benefits of using the Theseus execution engine.

The Theseus plan shown in Fig. 20 consists of eight operations for each recursive iteration of the plan. The Theseus execution engine executes several operations in parallel to reduce the response time of the composed service. For example, the operations 1a and 1b are executed in parallel as there is no dependency between them. Similarly, operations 2a and 2b and operations 4a and 4b are also executed in parallel. As a result of the optimized execution, the composed web service executes much more efficiently.

Furthermore, Theseus streams tuples between different operations as well. Imagine that operations 2a and 2b each provide 50 tuples in the first iteration. As soon as Theseus receives the first tuple from either operation, it passes the resulting tuple to the *Union* operation (operation 3). The *Union* operation passes the tuple to the *Minus* operation (4b) and the *Union* operation (4a). Streaming in Theseus results in more efficient execution.

6 Experimental results

In order to evaluate the different techniques described in this paper, we performed three sets of experiments with different real-world datasets. We wanted to test the following three hypotheses: (1) data integration techniques (as described in Sect. 3.3) can be used to generate composition plan for new web services from a large number of existing services, (2) the tuple-level filtering optimization algorithm described in Sect. 4 can be used to improve the execution time of the composition plans, and (3) the response time of the generated integration plan (and hence the response time of the composed service) can be reduced by translating the integration plan into a program that can be executed by a dataflow execution engine, such as Theseus as described in Sect. 5. All the experiments were performed on a PC running Windows XP with 1 GB of memory and 2.40 GHz of processor speed. The results are average of 10 runs.

6.1 Generating large composition plans

In this experiment, we show that the data integration techniques described in Sect. 3.3 can be used to generate plans for new web services that integrate data from a large number of existing services. To show this, we modeled a set of web services hosted by the National Cancer Institute (NCI) under a project called cancer Bioinformatics Infrastructure Objects (caBIO).[10] There are about 60 different web services each corresponding to different entities found in bioinformatics research, such as genes, proteins, or relationships between different entities. Moreover, all the web services can be accessed using different binding restrictions. For example, the web service to query information about a gene can be queried based on id, name, symbol, clusterid, or locuslinkid. We modeled all of those web services with different binding restrictions as data sources in the mediator model. The domain model consisted of 60 domain relations corresponding to 60 entity types and 496 data sources as each binding restriction resulted in one data source.

Once we modeled all the web services, we randomly generated requests to create different web services requiring the mediator to combine information about as many as 30 different entities by integrating information from as many as 60 web services (496 source descriptions). Figure 21 shows the composition time in milliseconds (ms) as we increase the complexity of the composed web service. As the results show, we can create a composition plan for a new web service that integrates information from 60 different web services in less than two seconds.

Executing the resulting composition plans using an efficient execution engine, such as Theseus, would typically

[10] http://ncicb.nci.nih.gov/core/caBIO.

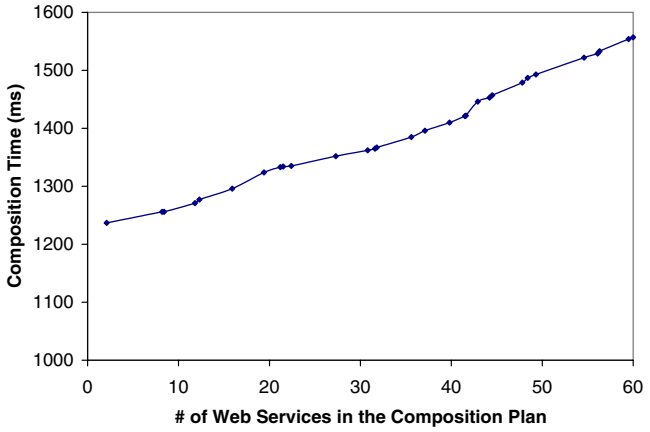

Fig. 21 Composing web service from large number of existing web services

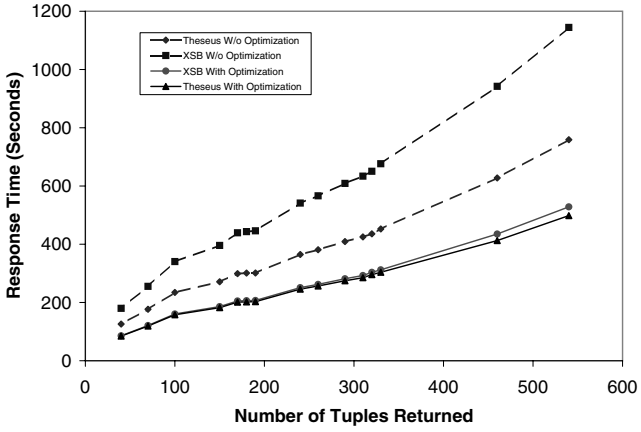

Fig. 22 Comparison of execution times of web service plans with and without the tuple-level filtering optimization

take over 30 s. The reason behind the large execution time is simply that to answer the given query the execution may need to send several requests to each of the existing web services and execution time of each request to a web service is between one and five seconds. Therefore, the composition time is much smaller than the execution time. This experiment supports the hypothesis that it is possible to utilize data integration techniques to compose new web services from a large number of existing web services.

6.2 Improvements resulting from tuple-level filtering

The goal of this set of experiments was to support the claim that by utilizing tuple-level filtering we can generate more efficient integration plans. In order to do this, we used the reformulation techniques described in Sect. 3.2 to generate the composition plan shown in Fig. 13(a). Then, we hosted a web service using the generated composition plan. Next, we used the tuple-level filtering algorithm to optimize the generated composition plan and hosted a separate web service with the optimized composition plan shown in Fig. 13(b).

As the *HSProteinInteractions* data source, we used the Database of Interaction Proteins (DIP) website.[11] We used Fetch AgentBuilder tools to wrap the website and convert it into a web service.[12] We wrapped and used the Biomolecular Interaction Network Database (BIND) as the *MMProteinInteractions* data source.[13]

In order to show that tuple-level filtering is useful regardless of the execution engine, we executed the generated integration plans using the Theseus execution engine and an open source Prolog interpreter termed XSB.[14] Since XSB does not have the ability to call web services, we used a Java program that accepted the input values for the *proteinid* and the *taxonid* attributes from the user, made necessary web service calls, and formatted the results of the web service calls as facts that were given as a part of the composition plan to XSB. As a result, we had four new web services: (1) composition plan without optimization executed with XSB, (2) composition plan without optimization executed with the Theseus execution engine, (3) composition plan with optimization executed with XSB, and (4) composition plan with optimization executed with the Theseus execution engine.

We sent requests to all new web services using different values of the *proteinid* and *taxonid* attribute. We measured the response time of all web services as the number of tuples returned by the composed web services increased. We divided all the proteins into groups based on number of protein–protein interactions, e.g. one group of proteins for which the composed web service returned between 15 and 30 tuples. There is a significant difference between the response time of the DIP and the BIND websites. Therefore, the response times of the composed web service varied depending on the organism of the input protein. To minimize the effect of the different response times, we randomly selected 5 human proteins and 5 mouse proteins from each group as inputs to all web services and averaged the response time of each web service over the 10 requests.

As shown in Fig. 22, the web service containing the optimized plan significantly outperformed the web service without the optimized plan regardless of which execution system we used. In fact, as the number of tuples returned by the composed service increased above 100 tuples, the web service executed with XSB with the optimized plan resulted in improvement of as much as 53.8% over the corresponding web service with unoptimized composition plans.

In this example web service, there is very little difference in the response time of the web service with optimized plan executed using XSB and the web service with optimized plan executed with Theseus. This is expected as the optimized web service implemented by the plan in Fig. 13(b) only requires one retrieval operation per recursive iteration,

[11] http://dip.doe-mbi.ucla.edu/dip/Search.cgi?SM=3.
[12] http://www.fetch.com.
[13] http://bind.ca/.
[14] XSB is available at http://xsb.sourceforge.net. Note that XSB is a prolog interpreter and can handle *Minus* operation.

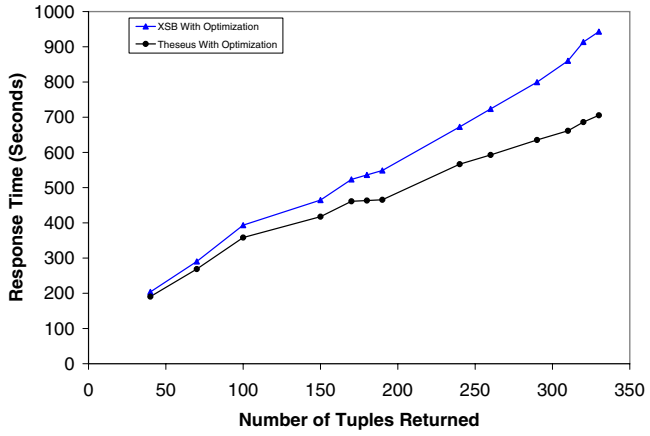

Fig. 23 Comparison of execution times of web service plans executed with XSB and Theseus

which means that no retrieval operations can be executed in parallel. However, Theseus is still able to stream tuples between operations. Therefore, the web service executing the optimized plan using Theseus outperforms the web service executing the optimized plan using XSB.

6.3 Improvements resulting from Theseus execution engine

Finally, our third set of experiments show that we can execute integration plans that require combining data from multiple data sources more efficiently by utilizing the Theseus execution engine to execute the plans instead of using a datalog evaluator such as XSB. Intuitively, Theseus can execute the integration plans more efficiently as it can stream tuples between different operations and execute multiple independent operations in parallel. While datalog evaluators are very efficient, they do not have the capability to execute multiple operations in parallel and stream tuples between different operations. In order to evaluate performance of Theseus compared to XSB, we hosted the optimized integration plan shown in Fig. 16 as a web service. The optimized composition plan for this web service may send as many as 10 web service requests per recursive iteration.

The data sources for the interactions were the same data sources we used in previous set of experiments, i.e. DIP and BIND. For the *HSProtein* and *MMProtein*, we wrapped the protein database provided by NCBI Entrez.[15] For the *TransducerProtein* and *MembraneProtein*, we wrapped the TigrFams website.[16]

We hosted two copies of the composed web service, one which used the Theseus execution engine to execute the composed web service and the second which utilized the XSB datalog interpreter with external Java program to request necessary data from the existing web services. We randomly picked several values of the *proteinid* attribute and sent requests to both web services. Figure 23 shows the comparison of the response times for both web services. As the number of tuples returned grew, the Theseus-based web service outperformed the XSB-based web service by as much as 33.6%. The improvement for the Theseus execution engine is due to the fact that Theseus can execute multiple web service requests in parallel and stream data between operations. As the number of requests to the existing web services increase, the improvement due to Theseus also increases. In this experiment, we only had access to two web services that provide information about proteins. As the number of available service increase, the improvement due to Theseus would be more pronounced.

[15] http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=Protein.
[16] http://www.tigr.org/TIGRFAMs/index.shtml.

7 Related work

The work presented in this article is closely related to research in several areas. The first area of related research is on data integration systems, such as the Information Manifold [12], InfoMaster [10], InfoSleuth [9], and Ariadne [11]. One can utilize any of the above-mentioned system to develop a framework for web service composition similar to what we have described in this paper. However, none of these systems were designed to work with parameterized queries. Therefore, they do not contain optimization techniques to optimize generalized composition plans that get generated from a parameterized query. One would need to augment their framework with techniques similar to the tuple-level filtering to handle generalized integration plans.

It should be noted that the tuple-level filtering algorithm can be used in conjunction with any mediator system that utilizes Local-As-View approach. In this article, we have used the Inverse Rules [13] algorithm to provide concrete example of how the optimization techniques and translation techniques would work. However, the techniques described in this paper are not specific to the Inverse Rules algorithm. The optimization and translation techniques described in this paper can be utilized with any system that utilizes Local-As-View [12, 16] model, i.e. we could use the Minicon algorithm [27] or the Bucket algorithm [14] instead of the Inverse Rules algorithm. However, the Minicon and Bucket algorithm cannot handle recursive queries. Therefore, if we use either of these algorithms, we cannot generate integration plans that require recursion. The tuple-level filtering algorithm would not be applicable if the data integration system utilizes the Global-As-View [15] approach as no source descriptions would be available in the Global-As-View model.

There has been work on streaming query execution in the data integration community [25, 26, 28, 29]. However, these streaming query execution engines have their own plan languages or rely on xquery and cannot directly execute datalog programs generated for web service composition.

The second area of relevant research is the data integration work applied to the problem of composing bioinformatics web services, such as BioMediator [30, 31], DiscoveryLink [18], BioKleisli [32, 33], TAMBIS [34, 35],

and Eckman et al.'s work [20, 36, 37]. While all the papers address different problems when integrating information from various bioinformatics data sources, the focus of these works is on (1) application of data integration techniques to answer specific queries by combining data from various bioinformatics data sources, (2) use cost-based optimization techniques to optimize the generated integration plan, and (3) support for a wide variety of binding patterns and access patterns. While we can support different binding patterns, we would need to utilize some of the techniques described in [37] to support different access patterns, such as queries on multi-valued attributes. One key difference between our work and the existing work is that we compose plans for hosting a web service. Therefore, we need to compose and optimize parameterized plans, as oppose to plans for specific queries. While the existing approaches can be modified to generate parameterized integration plans that can be hosted as web services, they would need to utilize optimization techniques similar to the tuple-level filtering technique described in this paper to do so efficiently.

The third area of related work is on optimization of data integration plans. In [19], 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 data accesses and to order access to different sources to reduce the query execution time in the 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.

In [38], we described the idea of using sensing operations to optimize data integration plans for specific user queries. 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. Moreover, in this paper we have extended the existing data integration techniques to generate parameterized integration plans that can be hosted as web services.

The fourth area of relevant research is on optimizing datalog programs. Kifer and Lozinskii [21] describe an approach to 'push' selections and projections in the datalog programs close to the sources. This is similar to tuple-level filtering without sensing operations. However, tuple-level filtering 'pushes' the selections that are in the source description. Those selections may or may not appear in the datalog program. Moreover, tuple-level filtering may insert sensing operations. There are many techniques in the literature to optimize datalog programs [39], such as magic sets. One can easily imagine an extension to our system that optimizes the generated datalog program using these techniques before translating a program to a Theseus plan.

Finally, there has been some research on automatic web service composition [40–42]. In particular, [42] describes an AI planning 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 [41], the authors use Golog [43] 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 reformulate different user queries. The focus of these papers is on composing web services and they do not address the issue of optimizing the execution of composed web services. The key advantages of their approaches is the ability to handle web services with world-altering preconditions and effects. However, the bioinformatics web services are largely information-providing services that our approach can handle efficiently.

8 Conclusion and future work

In this paper, we have described a mediator-based approach to automatically generate integration plans that can be hosted as web services. We showed that we can extend existing data integration techniques to generate integration plans for new web services. However, the generated plans are often inefficient. We described a novel optimization algorithm termed tuple-level filtering to optimize the integration plans using the order constraints in the source descriptions. The key new contribution of the tuple-level filtering algorithm is that unlike traditional optimization algorithms, the tuple-level filtering algorithm can be applied to parameterized integration plans and it may also insert sensing operations to improve the efficiency of the plans. Furthermore, we described techniques to translate recursive and non-recursive composition plans into integration plans that can be executed using a dataflow-style execution engine. Our experimental evaluation on real-world bioinformatics data sources shows that we can achieve a significant reduction in the response time of the composed web service using our optimization techniques.

We are working on extending our mediator to support a wide variety of operations on heterogeneous data. For example, dealing with complex XML structures or analyzing imagery data produced by different web services.

In addition, we are also looking into associating source descriptions with the composed web services for web service discovery purposes. Intuitively, the initial parameterized query used to generate the composition plan for the web service can be used as the source description of the composed service. Once we associate source descriptions to the composed web services, the mediator can utilize the composed web service automatically to create new more complex web services.

Acknowledgements We would like to thank Mark Carman, Louiqa Raschid, and the anonymous reviewers for their insightful comments that helped to improve the paper. This research is based upon work supported in part by the National Science Foundation under Award No. IIS-0324955, in part by the Defense Advanced Research Projects Agency (DARPA), through the Department of the Interior, NBC, Acquisition Services Division, under Contract No. NBCHD030010, in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command,

USAF, under agreement number F30602-00-1-0504, and in part by the Air Force Office of Scientific Research under grant number FA9550-04-1-0105. The US Government is authorized to reproduce and distribute reports for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors 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.

Appendix A Datalog to Theseus translation

A1. Datalog and Theseus plans with source access (Fig. 19a)

Figure 19(a) shows a datalog query that corresponds to retrieving *name*, *function*, *location* and *taxonid* information about the protein identified by *proteinid* '19456' from the *DIPProtein* data source.

```
Q(name,function, location, taxonid):-
       DIPProtein("19456", name, function,
          location, taxonid)
```

The corresponding Theseus plan is shown later. In Theseus, data flows through the program as relations. In the given example, the Theseus plan accepts one relation called *inrel* as input. The *inrel* relation is used to call a subplan termed *DIPProtein*. A *project* operation is used to project relevant attributes to the *outrel* stream, which is returned as output. The *DIPProtein* subplan, uses the *xwrapper* operation to retrieve XML data over HTTP. In the example plan, the XML data is retrieved from an agent that queries the information from the DIP website. As the XML data returned by the agent may contain lot of information, the *xquery* operation is used to extract the desired information. The resulting XML document looks like XML representation of a relation. This XML document is converted to a relation using the *xml2rel* operation. Finally, a *project* operation is used to project the necessary attributes to the *outrel*.

```
RELATION inrel:id char
19456

PLAN Q
{
  INPUT: stream inrel
  OUTPUT: stream outrel
  BODY
  {
    DIPProtein(inrel : dipout)
    project(dipout, "name, function, location,
      taxonid" : outrel)
  }
}
PLAN DIPProtein
{
  INPUT: stream inrel
  OUTPUT: stream outrel
  BODY
  {
    xwrapper("http://localhost:8080/agent/
      runner?plan=
               DIPProtein/plans/production",
                "nodeid=id",
              inrel, "wrapper_data" :
                wrapperout)
    xquery(wrapperout, "wrapper_data",
           "<proteins>
             {for $b in input()//Document//Row
                return <protein>
                 <name>{$b/name/text()}</name>
                 <function>{$b/function/text()}
                   </function>
                 <location>{$b/location/text()}
                   </location>
                 <taxonid>{$b/taxonid/text()}
                   </taxonid>
              </protein> }
            </proteins>", "answer" : xqueryout)
    xml2rel(xqueryout, "answer", "/proteins/
      protein", "index" : x2ro)
    project(x2ro,"proteinid, name, function,
      location, taxonid"
               : outrel)
  }
}
```

Datalog and Theseus plans with a select (Fig. 19b)

Figure 19b shows a datalog query that corresponds to retrieving *name*, *function*, *location* and *taxonid* information about the protein identified by *proteinid* '19456' from the *DIPProtein* data source and performing a selection based on *taxonid* attribute.

```
Q(name, function, location, taxonid):-
       DIPProtein("19456", name, function,
        location, taxonid) ^
       taxonid > 9600
```

Below is the corresponding Theseus plan. The Theseus plan accepts one relation called *indata*. The input stream is used to call a subplan termed *DIPProtein*. A *select* operation is used to filter out tuples that with value of *taxonid* attribute less than or equal to 9600. A *project* operation is used to project relevant attributes to the *outdata* stream, which is returned as output. The *DIPProtein* subplan is the same as the *DIPProtein* plan described in Appendix 1.1.

```
RELATION indata:id char
19456

PLAN Q
{
  INPUT: stream indata
  OUTPUT: stream outdata
  BODY
  {
    DIPProtein(indata : dipproteinout)
    select(dipproteinout, "taxonid > 9600"
      : selectout)
    project(selectout, "name, function,
      location, taxonid" : outdata)
  }
}
```

Datalog and Theseus plans with a join (Fig. 19c)

Figure 19c shows a datalog query with a join between two relations.

```
Q(paperid, conf, year, loc):-
       Papers(paperid, conf, year) ^
       Conference(conf, year, loc)
```

Below is the corresponding Theseus plan. In this plan, Theseus executes both *papers* and *conference* subplans in parallel as there is no dependency between them. A *join* operation is used to combine information from both plans. A *project* operation is used to project relevant attributes to the *outdata* stream, which is returned as output. The subplans for the *Papers* and *Conference* data sources are not shown as they are very similar to the *DIPProtein* plan shown in Appendix 1.1.

```
PLAN Q
{
  INPUT:
  OUTPUT: stream outdata
  BODY
  {
    papers( : papersout)
    conference( : conferenceout)
    join(papersout, conferenceout, "l.conf=
      r.conf and l.year=r.year"
            : jout)
    project(jout, "paperid, conf, year, loc" :
      outdata)
  }
}
```

Datalog and Theseus plans with a dependent join (Fig. 19(d))

Figure 19d shows a datalog query that results in a dependency between operations due to binding restrictions.

```
Q(id, conf, year, author, email):-
       Papers(id, conf, year) ^
       PaperDetails($id, author, year,
        institute, email)
```

Below is the corresponding Theseus plan. In this plan, there is a dependency between the call to *papers* and the call to *paperdetails* subplans. Therefore, Theseus first executes the *papers* subplan. As soon as the *papers* subplan returns the first tuple, it is used to call *paperdetails* subplan. A *join* operation is used to combine information from both plans. A *project* operation is used to project relevant attributes to the *outdata* stream, which is returned as output.

```
PLAN Q
{
  INPUT:
  OUTPUT: stream outdata
  BODY
  {
    papers( : papersout)
    project(papersout, "id" : paperdetailsin)
    paperdetails( paperdetailsin :
      paperdetailsout)
    join(papersout, paperdetailsout, "l.id=
      r.id and l.year=r.year"
            : jout)
    project(jout, "id, conf, year, author,
      email" : outdata)
  }
}
```

A.2 Datalog and Theseus plans with a union (Fig. 19e)

Figure 19d shows a datalog query that contains a union operation.

```
Q(fromid, toid, source):-
    HSProteinInteractions(fromid, toid, source,
      verified) ^
    fromid = 19456
Q(fromid, toid, source):-
    MMProteinInteractions(fromid, toid, source,
      verified) ^
    fromid = 19456
```

Below is the corresponding Theseus plan. In this plan, Theseus executes both *HSProteinInteractions* and *MMProteinInteractions* subplans in parallel as there is no dependency between them. A *union* operation is used to combine information from both plans. A *project* operation is used to project relevant attributes to the outdata stream, which is returned as output.

```
RELATION indata:fromid char
19456

PLAN Q
{
  INPUT: stream indata
  OUTPUT: stream outdata
  BODY
  {
    HSProteinInteractions(indata : HSout)
    MMProteinInteractions(indata : MMout)
    union(HSout, MMout : uout)
    project(uout, "fromid, toid, source" :
      outdata)
  }
}
```

A3. Example Theseus plan with recursion (Fig. 20)

Below is the Theseus plan for the recursive integration plan shown in Fig. 20. Note that the last statement of the plan calls itself back denoting recursion. The plan accepts two relations as input: one relation called *indata* containing *proteinid* and *taxonid* and the other called *InteractionsSoFar* containing *proteinid* and *toid*. The *select* operator is used to determine which tuples should be passed as inputs to the *HSProteinInteractions* and *MMProteinInteractions*. The output from both services is unioned to generate all interactions found in the current iteration. The resulting relation is unioned with the *InteractionsSoFar* relation to obtain interactions found in all iterations so far. In parallel, Theseus performs a *minus* operation between the interactions found in the current iterations and *InteractionsSoFar* relation to obtain all new interactions found in the current relation. A *null* operator is used to check if any new interactions were found in this iteration. If no new interactions were found, then Theseus passes the relation containing all interactions seen so far as the output. Otherwise, it passes a relation containing new interactions found in the current iteration and a relation containing all interactions seen so far as the input to the next recursive iteration.

```
RELATION indata:proteinid char, taxonid char
105096|9606
110596|10090
RELATION InteractionsSoFar:proteinid char,
  toid char

PLAN InteractionsPlan
{
  INPUT:stream indata, stream InteractionsSoFar
  OUTPUT:stream outdata
  BODY
  {
    select(indata, "taxonid='9606'":
      select1out)
    select(indata, "taxonid='10090'":
      select2out)
    HSProteinInteractions(select1out:HSout)
    MMProteinInteractions(select2out:MMout)
    union(HSout, MMout: Current)
    union(InteractionsSoFar, Current : Newout)
    minus(Current, InteractionsSoFar : Newin)
    null(Newin, Newout, Newin : outdata,
```

```
      nextin)
    project(nextin, "toid, taxonid": prjnextin)
    InteractionsPlan(prjnextin, newout :
      outdata)
  }
}
```

References

1. Bright, L., Gruser, J.-R., Raschid, L., Vidal, M.E.: A wrapper generation toolkit to specify and construct wrappers for web accessible data sources (web sources). J. Comput. Syst. Sci. Eng. **14**(2), (1999)
2. Kushmerick, N., Weld, D., Doorenbos, R.: Wrapper induction for information extraction. In: Proceedings of the International Conference on Artificial Intelligence, IJCAI-97 (1997)
3. Muslea, I., Minton, S., Knoblock, C.A.: Selective sampling with redundant views. In: Proceedings of the 17th National Conference on Artificial Intelligence (2000)
4. Schoppers, M.: Universal plans for reactive robots in unpredictable environments. In: Proceedings of the International Conference on Artificial Intelligence, IJCAI-87 (1987)
5. Thakkar, S., Ambite, J.L., Knoblock, C.A.: A view integration approach to dynamic composition of web services. In: Proceedings of 2003 ICAPS Workshop on Planning for Web Services. Trento, Italy (2003)
6. Thakkar, S., Ambite, J.L., Knoblock, C.A.: A data integration approach to automatically composing and optimizing web services. In: Proceedings of 2004 ICAPS Workshop on Planning and Scheduling for Web and Grid Services (2004)
7. Thakkar, S., Knoblock, C.A.: Efficient execution of recursive integration plans. In: Proceeding of 2003 IJCAI Workshop on Information Integration on the Web. Acapulco, Mexico (2003)
8. Tejada, S., Knoblock, C.A., Minton, S.: Learning domain-independent string transformation weights for high accuracy object identification. In: Proceedings of the Eighth ACM SIGKDD International Conference. Edmonton, Alberta, Canada (2002)
9. Bayardo, R.J., Jr., Bohrer, W., Brice, R.S., Cichocki, A., Flower, J., Helal, A., Kashyap, V., Ksiezyk, T., Martin, G., Nodine, M., Rashid, M., Rusinkiewicz, M., Shea, R., Unnikrishnan, C., Unruh, A., Woelk, D.: Infosleuth: agent-based semantic integration of information in open and dynamic environments. In: Proceedings of ACM SIGMOD-97 (1997)
10. Genesereth, M.R., Keller, A.M., Duschka, O.M.: Infomaster: an information integration system. In: Proceedings of ACM SIGMOD-97 (1997)
11. Knoblock, C.A., Minton, S., Ambite, J.-L., Ashish, N., Muslea, I., Philpot, A., Tejada, S.: The ariadne approach to web-based information integration. Int. J. Intell. Cooperative Inform. Syst. (IJCIS) **10**(1–2), 145–169 (2001)
12. Levy, A.Y., Rajaraman, A., Ordille, J.J.: Query-answering algorithms for information agents. In: Proceedings of AAAI-96 (1996)
13. Duschka, O.M.: Query planning and optimization in information integration. PhD thesis, Stanford University (1997)
14. integration. In: Minker, J. (ed.) Logic Based Artificial Intelligence. Kluwer, Boston (2000)
15. Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., Widom, J.: Integrating and accessing heterogeneous information sources in tsimmis. In: Proceedings of the AAAI Symposium on Information Gathering. Stanford, CA (1995)
16. Lenzerini, M.: Data integration: a theoretical perspective. In: Proceedings of ACM Symposium on Principles of Database Systems. Madison, WI, USA (2002)
17. Golden, K.: Leap before you look: information gathering in the puccini planner. In: Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems (1998)
18. Haas, L.M., Kodali, P., Rice, J.E., Schwarz, P.M., Swope, W.C.: Integrating life sciences data-with a little garlic. In: Proceedings of the IEEE International Symposium on Bio-Informatics and Biomedical Engineering (BIBE'00), pp. 5–13 (2000)
19. Kambhampati, S., Lambrecht, E., Nambiar, U., Nie, Z., Gnanaprakasam, S.: Optimizing recursive information gathering plans in emerac. J. Intell. Inform. Syst. (2003)
20. Lacroix, Z., Raschid, L., Eckman, B.A.: Techniques for optimization of queries on integrated biological resources. J. Bioinform. Comput. Biol. **2**(2), 375–411 (2004)
21. Kifer, M., Lozinskii, E.L.: On compile-time query optimization in deductive databases by means of static filtering. ACM Trans. Database Syst. **15**(3), 385–426 (1990)
22. Levy, A.Y., Suciu, D.: Deciding containment for queries with complex objects. In: Proceedings of the 16th ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems, pp. 20–31 (1997)
23. Lacroix, Z., Raschid, L.: A map of biological resources to support a complete characterization of scientific entities. Technical report, University of Maryland (2002)
24. Michalowski, M., Thakkar, S., Knoblock, C.: Automatically utilizing secondary sources to align information across sources, special issue on semantic integration. AI Mag. **26**(1), 33–45 (2005)
25. Ives, Z.G., Florescu, D., Friedman, M., Levy, A., Weld, D.S.: An adaptive query execution system for data integration. In: ACM SIGMOD Conference (1999)
26. Barish, G., Knoblock, C.A.: An expressive language and efficient execution system for software agents. J. Artif. Intell. Res. **23**, 625–666 (2005)
27. Pottinger, R., Levy, A.: A scalable algorithm for answering queries using views. VLDB J. 484–495 (2000)
28. Hellerstein, J.M., Franklin, M.J., Chandrasekaran, S., Deshpande, A., Hildrum, K., Madden, S., Raman, V., Shah, M.A.: Adaptive query processing: technology in evolution. IEEE Data Eng. Bull. **23**(2), 7–18 (2000)
29. Naughton, J.F., DeWitt, D.J., Maier, D., Aboulnaga, A., Chen, J., Galanis, L., Kang, J., Krishnamurthy, R., Luo, Q., Prakash, N., Ramamurthy, R., Shanmugasundaram, J., Tian, F., Tufte, K., Viglas, S., Wang, Y., Zhang, C., Jackson, B., Gupta, A., Chen, R.: The niagara Internet query system. IEEE Data Eng. Bull. **24**(2), 27–33 (2001)
30. Mork, P., Halevy, A., Tarczy-Hornoch, P.: A model for data integration systems of biomedical data applied to online genetic databases. In: Proceedings of the American Medical Informatics Association Fall Symposium (AMIA) (2001)
31. Mork, P., Shaker, R., Halevy, A., Tarczy-Hornoch, P.: Pql: a declarative query language over dynamic biological schemata. In: Proceedings of the American Medical Informatics Association Fall Symposium (AMIA). San Antonio, TX (2002)
32. Buneman, P., Crabtree, J., Davidson, S.B., Overton, C., Tannen, V., Wong, L., BioKleisli: Integrating biomedical data and analysis packages. In: Letovsky, S. (ed.) Bioinformatics: Databases and Systems. Kluwer Academic Publishers, pp. 201–217 (1999)
33. Davidson, S.B., Overton, G.C., Tannen, V., Wong, L.: Biokleisli: a digital library for biomedical researchers. Int. J. Digital Libraries **1**(1), 36–53 (1997)
34. Goble, C.A., Stevens, R., Ng, G., Bechhofer, S., Paton, N.W., Baker, P.G., Peim, M., Brass, A.: Transparent access to multiple bioinformatics information sources, special issue on deep computing for the life sciences. IBM Syst. J. **40**(2), 532–552 (2001)
35. Stevens, R., Goble, C., Paton, N.W., Bechhofer, S., Ng, G., Baker, P., Brass, A.: Complex query formulation over diverse information sources in TAMBIS. In: Lacroix, Z., Critchlow, T. (eds.) Bioinformatics: Managing Scientific Data. Morgan Kaufmann, San Francisco, CA (2003)
36. Eckman, B.A., Kosky, A.S., Laroco, L.A., Jr.: Extending traditional query-based integration approaches for functional characterization of post-genomic data. Bioinformatics **17**(7), 587–601 (2001)

37. Eckman, B.A., Lacroix, Z., Raschid, L.: Optimized seamless integration of biomolecular data. In: Proceedings of the 2nd IEEE International Symposium on Bioinformatics and Bioengineering (BIBE'01), pp. 23–32 (2001)
38. Ashish, N., Knoblock, C.A., Levy, A.: Information gathering plans with sensing actions. In: European Conference on Planning, ECP-97. Toulouse, France (1997)
39. Ullman, J.: Principles of Data and Knowledge-Base Systems. Computer Science Press, New York (1988)
40. Bultan, T., Fu, X., Hull, R., Su, J.: Conversation specification: a new approach to design and analysis of e-service composition. In: Proceedings of 12th International World Wide Web Conference (WWW) (2003)
41. McIlraith, S., Son, T.C.: Adapting golog for composition of semantic web services. In: Proceedings of the 8th International Conference on Knowledge Representation and Reasoning (KR'02). Toulouse, France (2002)
42. Wu, D., Parsia, B., Sirin, E., Hendler, J., Nau, D.: Automating daml-s web services composition using shop2. In: 2nd International Semantic Web Conference (ISWC2003) (2003)
43. Levesque, H.J., Reiter, R., Lesperance, Y., Lin, F., Scherl, R.B.: GOLOG: a logic programming language for dynamic domains. J. Logic Program. **31**(1–3), 59–83 (1997)