SUSIE: Search Using Services and Information Extraction

Nicoleta Preda[1], Fabian Suchanek[2], Wenjun Yuan[3], Gerhard Weikum[2]

[1] University of Versailles, France, npreda@prism.uvsq.fr

[2] Max Planck Institute for Informatics, Germany, {suchanek,weikum}@mpi-inf.mpg.de

[3] Hong Kong University, China, wjyuan@cs.hku.hk

***Abstract*— The API of a Web service restricts the types of queries that the service can answer. For example, a Web service might provide a method that returns the songs of a given singer, but it might not provide a method that returns the singers of a given song. If the user asks for the singer of some specific song, then the Web service cannot be called – even though the underlying database might have the desired piece of information. This asymmetry is particularly problematic if the service is used in a Web service orchestration system.**

In this paper, we propose to use on-the-fly information extraction to collect values that can be used as parameter bindings for the Web service. We show how this idea can be integrated into a Web service orchestration system. Our approach is fully implemented in a prototype called SUSIE. We present experiments with real-life data and services to demonstrate the practical viability and good performance of our approach.

I. Introduction

A. Motivation

There is a growing number of Web services that provide a wealth of information. There are Web services about books (*isbndb.org*, *librarything. com*, *Amazon*, *AbeBooks*), about movies (*api.internetvideoarchive.com*), about music (*musicbrainz.org*, *lastfm.com*), and about a large variety of other topics. Usually, a Web service is an interface that provides access to an encapsulated back-end database. For example, the site *musicbrainz.org* offers a Web service for accessing its database about music albums. The Web service defines functions that can be called remotely. *musicbrainz.org* offers the function *getSongs*, which takes a singer as input parameter and delivers the songs by that singer as output. If the user wants to know all songs by Leonard Cohen, she can call *getSongs* with Leonard Cohen as input. The output will contain the songs *Suzanne*, *Hallelujah*, etc.

Web services play a crucial part in the trend towards data-centric applications on the Web. Unlike Web search engines, Web services deliver crisp answers to queries. This allows the user to retrieve answers to a query without having to read through several result pages. Web services can also be used to answer precise conjunctive queries, which would require several searches on the Web and joins across them, if done manually with a search engine. The results of Web services are machine-readable, which allows query answering systems to cater to complex user demands by orchestrating the services. These are advantages that Web services offer over keyword-based Web search.

Web services allow querying remote databases. However, the queries have to follow the *binding patterns* of the Web service functions, by providing values for mandatory input parameters before the function can be called. In our example of musicbrainz, the function *getSongs* can only be called if a singer is provided. Thus, it is possible to ask for the songs of a given singer, but it is not possible to ask for the singers of a given song. If the user wants to know, e.g., who sang *Hallelujah*, then the Web service cannot be used to answer this question – even though the database contains the desired information. This restriction is not due to missing data, but a design choice of the Web service owner, who wants to prevent external users from extracting and downloading large fractions of its back-end database. On the client side, this is a highly inconvenient limitation, in which the data may be available, but cannot be queried in the desired way. We call this the problem of *Web service asymmetry*.

Intuitively, a binary relation R is asymmetric with respect to a set of functions, if the functions allow querying for one argument of R but not for the other one. Even among the most prominent data-service providers, many relations are asymmetric. We have examined *isbndb.org*, *librarything.com*, and *abebooks.com* for books, *internetvideoarchive.com* for movies, *musicbrainz.org*, *last.fm*, *discogs.com*, and *lyricWiki.org* for music. Table 1 lists relations that can be queried for the second argument, but not for the first.

citizenOf(pers,country)	*rating(movie,x)*
bornIn(pers,year)	*graduatedFrom(pers,univ)*
livesIn(pers,place)	*published(book,year)*
hasWon(pers,award)	*publishedBy(book,editor)*

Fig. 1. Asymmetric relations in Web services.

The asymmetric relations are by no means outlandish: It is legitimate, e.g., to ask for singers who won a certain prize. The only way to deal with such asymmetric Web services is to try out all possible input values until the Web service delivers the desired output value. For example, if the user asks for the singer of the song *Hallelujah*, then we can use a semantic knowledge base such as YAGO [1], Freebase[1] or DBpedia [2] to get a list of singers. Then we call *getSongs* with every singer, and remember those for which the Web service returns *Hallelujah*. Obviously, this approach quickly

[1] *http://freebase.com*

becomes infeasible. The first limitation is runtime, with Web service calls taking up to 1 second to complete. Trying out thousands of singers that a knowledge base such as YAGO contains could easily take hours. The second limitation is the data provider itself, which most likely restricts aggressive querying from the same IP address. If the asymmetric relations could be queried on a case-by-case basis, without materializing the entire function, then both the user and the provider would be helped: The user, because she can answer her queries; and the provider, because aggressive querying is avoided.

The functions exported by Web service APIs can be seen as views with binding patterns [3]. There are several methods to evaluate queries on such views efficiently [3–10]. Yet, these approaches do not address the asymmetry issue. If faced with an asymmetric relation, these approaches have to enumerate an entire domain until they stumble upon the correct value. Since the approaches assume an infinite budget of function calls, they can afford to enumerate the domain. In the context of Web services, however, this approach is out of question.

The problem is even more challenging, because asymmetric relations may appear in query plans that orchestrate several Web service functions. Assume, e.g., that the user wishes to find the albums that feature the song *Hallelujah*. Assume that we have a service that delivers the album of a song, if we provide the id of the song. To retrieve the id of the song, we have to call another function that requires the singer of the song. This gives rise to a query composition tree where the asymmetric relation is a leaf node. If an alternative service composition also promises to deliver the album, then both compositions have to be weighted against each other. This is not trivial. Assume, e.g., a service that, given a song, delivers songs that it inspired, together with their album. A service orchestration system might call this service with *Hallelujah*, retrieve inspired songs, and then call the service again with these results, and again and again, in the vain hope to find the album of *Hallelujah*. This way, it descends into a chaining of derived songs. We show in this paper that, even if the inverse functions are available, standard approaches to query answering cannot prioritize them over the chains.

B. Contribution

In this paper, we develop a solution to the problem of Web service asymmetry. We propose to use Web-based information extraction (IE) on the fly to determine the right input values for the asymmetric Web services. For example, to find all singers of *Hallelujah*, we issue a keyword query "singers Hallelujah" to a search engine. We extract promising candidates from the result pages, say, *Leonard Cohen*, *Lady Gaga*, and *Elvis*. Next, we use the existing Web service to validate these candidates. In the example, we would call *getSongs* for every candidate, and see whether the result contains *Hallelujah*. This confirms the first singer and discards the others. This way, we can use an asymmetric Web service as if it allowed querying for an argument that its API does not support. We show how such functions can be integrated into a Web orchestration system, and how they can be prioritized over infinite chains of calls. Our technique works only if the Web provides adequate candidate entities. We show in our experiments that this is the case in an interesting spectrum of applications. Our paper makes the following contributions:

1) A solution to the problem of Web service asymmetry, where input values for the Web services are extracted on-the-fly from Web pages found by keyword queries.
2) A modification to the standard Datalog evaluation procedure that prioritizes inverse functions over infinite call chains.
3) An experimental evaluation of our approach with the APIs of real Web services, showing how we can improve the performance of a real-world query answering system.

Our methods are fully implemented in the SUSIE system (Search Using Services and Information Extraction). The rest of this paper is structured as follows. Section II contains preliminaries. Section III discusses execution plans. Section IV introduces the key concept of inverse functions and their scheduling. Section V discusses the generation of inverse functions and their implementation. Section VI presents comprehensive experiments with real-world Web services. Section VII discusses related work, and we conclude in Section VIII.

II. Preliminaries

Global Schema. A Web service defines an API of functions that can be called over the Internet. Given some input values, a function returns as output a semi-structured document, usually in XML. In all of the following, we assume that the set of Web service functions is known. We also assume that all functions operate on the same, conceptually global database with a unified schema. This is an important assumption in our context, which we make in line with other works in the area [3–8, 11, 12]. The mappings from the actual schemas of the functions to the global schema can be defined manually, or they can be automatically generated using tools [13]. We see this as a challenge that is orthogonal to the present work. The fact that the functions operate on different data sources can be modeled by source designators, as we shall see later. As a running example, consider the global schema and the database shown in Figure 2.

sang

singer	songId
Cohen	1
Elvis	2
Elvis	3

hasTitle

songId	title
1	Hallelujah
2	All Shook Up
3	Teddy Bear

onAlbum

songId	album
1	Various Positions
2	Memphis To Vegas
3	Loving You

Fig. 2. A sample global database

Function Definitions. Users and application programs do not have access to the global database. Rather, access has to go through the functions provided by Web services operating on the global database. For example, a function can be *getSongs(in:singer, out:songId, out:title)*. This function expects singer as input and returns the song id and title as output. In all of the following, i, t, a, and s will be variables for song ids, song titles, albums, and singers, respectively. If we call *getSongs(Cohen, i, t)*, we receive as output *i=1*, *t=Hallelujah*. Seen this way, the function provides a view on the global database. More formally, and again in line with [3–8, 11, 12], we see the Web service functions as views with binding patterns over the global database.

Definition 1 (Function Definition): A function definition over a global database schema is a rule of the form

$$f^{\bar{A}}(\bar{X_0}) \leftarrow r_1(\bar{X_1}), r_2(\bar{X_2}), \ldots r_n(\bar{X_n})$$

where $r_1, r_2, \ldots r_n$ are database relations and the $\bar{X_i}$ are tuples of variables and/or constants. All variables of the head must occur in the body of the rule. $\bar{A}$ is the adornment of the function. It is a string of length $|\bar{X_0}|$ composed of the letters b and f. The meaning of the adornment b is that a binding value must be provided for the variable in that position, whereas the adornment f does not impose such a restriction.
The function *getSongs*, e.g., can be written as follows:

getSongs^{bff}*(s,i,t)* ←*hasTitle(i,t), sang(s,i)*

This rule defines how the Web service provider computes the function *getSongs*. From the caller's point of view, the adornment *bff* states that *getSongs* can be called only if a value for s is provided. The evaluation of the function call binds the variables i and t to the values that satisfy the conditions *hasTitle(i,t)* and *sang(s,i)*. Variables that appear in the body of the rule, but not in its head are called *existential variables*. For example, a function may return all albums by a singer:

getAlbums^{bf}*(s,a)* ←*sang(s,i), onAlbum(i,a)*

Again, this rule specifies how the Web service provider computes the function. These functions are the only way for a user or application to access the global database.

INCOMPLETE FUNCTIONS. Web services are often incomplete. Suppose, e.g., that *musicbrainz* contains only Elvis. We model this incompleteness by source designators. A source designator is an atom that can be appended to the body of a function to indicate that the underlying data source is incomplete. In the example, we define

getAlbums$^{bf}_{mb}$*(s,a)*← *sang(s,i), onAlbum(i,a), mb(s)*

where *mb(s)* is a global relation that holds for those singers that *musicbrainz* knows. Appropriate source designators can model the fact that some functions of the same Web service talk about the same entities, or conversely, that functions of different services operate on different data sources.

DATABASE FUNCTIONS. In some cases, the user has access to a database or knowledge base such as YAGO [1] or DBpedia [2]. These databases can also be seen as (incomplete) functions on the global database. They have only f-adornments and no b-adornments, and whenever they are called, they are instantiated with tuples from the database. The following function, e.g., is a database function that delivers all singers from YAGO:

getSingers$^{f}_{yago}$*(s)*← *sang(s,i), yago(s)*

QUERIES. We consider the evaluation of *conjunctive queries* defined over the global schema. For uniformity and in line with [3, 11] we adopt the Datalog notation for queries.

Definition 2 (Query): A query is a datalog rule of the form

$$q(\bar{X_0}) \leftarrow r_1(\bar{X_1}), r_2(\bar{X_2}), \ldots r_n(\bar{X_n})$$

where $r_1, r_2, \ldots r_n$ are database relations and the $\bar{X_i}$ are tuples of variables and/or constants. All variables of the head must occur in the body of the rule.
The following query, e.g., asks for the singer of *Hallelujah*:

q(s) ←*hasTitle(i, Hallelujah), sang(s,i)*

Variables that appear in the body of the rule, but not in its head are called *existential variables*. An *answer to a query* is an answer to the query on the global database. In the example, an answer to the query is *s=Cohen*.

III. EXECUTION PLANS

GOAL. Our goal is to answer queries by using only function calls. This goal is in line with the works [3, 7, 8]. The difficulty in answering queries lies in the fact that the query is formulated in terms of the global schema, not in terms of the functions. This is because the query expresses an information need, and not yet a constructive procedure. Since the user does not have access to the global database, we cannot execute the query directly on the tables of the global database. Rather, the algorithms automatically translate the query into query plans expressed in terms of the available Web service functions, respecting their binding pattern restrictions. This is a non-trivial endeavor that we discuss next.

Different from previous work, we consider a given budget of calls. This changes the goal of the evaluation. Previous work aimed to compute the maximal number of query answers. The goal was to compute the *maximal contained rewriting*. Unfortunately, when the views have binding pattens, even the evaluation of conjunctive queries requires rewritings of unbound length [3]. Thus, these works will produce pipelines of calls of an unbound length in order to obtain the maximal number of answers. This is infeasible in the context of Web services. Our goal, in contrast, is to compute the largest number of answers using the given budget of calls. Hence, we have to prioritize calls that are likely to lead to an answer. This is different from the problem of join ordering in previous work. Therefore, we have to use a different model for the query answering process: our execution plans are ordered sequences of calls rather than ordered join plans.

EXECUTION PLANS. Consider the following query, which asks for albums by Cohen:

q_1(a) ←*sang(Cohen,i), onAlbum(i,a)*

We cannot access the tables *sang* and *onAlbum* directly. Rather, we have to access the tables through functions. Suppose, e.g., that we have the following functions:

getAlbum^{bf}*(i,a)* ←*onAlbum(i,a)*
getSongs^{bff}*(s,i,t)* ←*sang(s,i), hasTitle(i,t)*

We can first call *getSongs*^{bff}*(Cohen, i, t)*, which will return the ids and titles of all songs by Cohen (i.e., *i*=1, *t=Hallelujah*). Then, we call *getAlbum*^{bff}*(i,a)* with every binding of i, which will give us values for a. In the example, there is only one value, *a=VariousPositions*. The sequence of calls is:

$$\textit{getSongs}^{bff}\textit{(Cohen, i, t), getAlbum}^{bff}\textit{(i,a)}$$

This is a valid execution plan, because every b-argument of a function will have a value at execution time. We make this notion more formal now.

Definition 3 (Function Call): A function call of a function f is a literal of the form $f(\bar{X})$, where $\bar{X}$ is a sequence of variables and constants of the same arity as f.
In the example, *getSongs*^{bff}*(Cohen,i,t)* and *getAlbum*^{bff}*(i,t,a)* are function calls.

Definition 4 (Consequences): The consequences of a function call $f(\bar{X})$ are the body atoms of the function definition of f, with variables substituted by the values from $\bar{X}$. Existential variables of f are substituted by fresh variables in the consequences.
In the example, the consequences of *getSongs*^{bff}*(Cohen,i,t)* are $\{$*hasTitle(i,t), sang(Cohen,i)*$\}$.

Definition 5 (Execution Plan): An execution plan for a query Q is a sequence of function calls. Each argument in the execution plan has to be either (1) a constant symbol that appears in Q or (2) a variable.
The plan is *admissible* if, for every variable, the first occurrence of the variable in the call sequence is in an f-position of a function call. The plan is *effective* if all body atoms of Q appear in the conjunction of the function call consequences (where existential variables of Q can be matched with any variables or constants in the consequences). Let us look again at our sample plan above. The plan is admissible because every b-argument of a function call is either a constant or has been bound by a previous function call. The plan is effective because the query atoms *sang(Cohen, i)*, and *onAlbum(i,a)* appear in the conjunction of the consequences of the calls:

sang(Cohen, i), hasTitle(i,t), onAlbum(i,a)

GUESSING PLANS. Not all plans are promising. Assume, e.g., that the user asks for the album and singer of *Hallelujah*:

$$q_2\textit{(a,s)} \leftarrow \textit{hasTitle(i, Hallelujah), onAlbum(i,a), sang(s,i)}$$

Assume that we have the following functions:

$$\textit{getSongInfo}^{bfff}\textit{(i,t,s,a)} \leftarrow \textit{hasTitle(i,t),sang(s,i),onAlbum(i,a)}$$
$$\textit{getRelSongs}^{bff}(t_1, i_2, t_2) \leftarrow \textit{influenced}(i_1, i_2), \textit{hasTitle}(i_1, t_1), \textit{hasTitle}(i_2, t_2)$$

The first function returns all information about a song id. The second function returns the id and title of a song that was influenced by the input song title. A similar function is published by *musicbrainz*. This allows for the following plan:

$$\textit{getRelSongs}^{bff}\textit{(Hallelujah,}i_2, t_2\textit{)}, \textit{getSongInfo}^{bfff}\textit{(}i_2\textit{,Hallelujah,s,a)}$$

This plan starts with the song *Hallelujah*, and finds which songs it influenced. Each of these songs comes with an id i_2 and a title t_2. Then the plan calls *getSongInfo*^{bfff} to check whether i_2 was by any chance the id of *Hallelujah*, in which case we get the singer s and the album a. Obviously, this is unlikely to succeed in reality. We call such a plan a *guessing plan* (a notion that we will make more precise later).

UNBOUND GUESSING PLANS. In some cases, there are infinitely many guessing plans. Our query q_2, e.g., gives rise to the following unbound number of plans:

$$\textit{getRelSongs}^{bff}\textit{(Hallelujah,}i_2, t_2\textit{)},$$
$$\textit{getRelSongs}^{bff}(i_2, i_3, t_3), \ldots, \textit{getRelSongs}^{bff}(i_{n-1}, i_n, t_n),$$
$$\textit{getSongInfo}^{bfff}(i_n\textit{,Hallelujah,s,a)}$$

These plans will enumerate the entire domain of songs, in order to "guess" the input value of *getSongInfo*^{bfff}. In general, there can be infinitely many pipelines for a given query under a given set of functions [14]. We call such plans *unbound plans*. The goal of SUSIE is to avoid guessing plans and unbound guessing plans by adding *inverse functions*.

IV. EXECUTION PLANS WITH INVERSE FUNCTIONS

This section will introduce the core contribution of SUSIE, inverse functions. We will show how to prioritize them over guessing plans.

A. Adding Inverse Functions

Let us consider again the query for the singer and album of *Hallelujah*:

$$q_2\textit{(a,s)} \leftarrow \textit{hasTitle(i, Hallelujah),onAlbum(i,a),sang(s,i)}$$

As before, we have the following functions:

$$\textit{getSongInfo}^{bfff}\textit{(i,t,s,a)} \leftarrow \textit{hasTitle(i,t),sang(s,i),onAlbum(i,a)}$$
$$\textit{getSongs}^{bff}\textit{(s,i,t)} \leftarrow \textit{sang(s,i), hasTitle(i,t)}$$
$$\textit{getAlbum}^{bf}\textit{(i,a)} \leftarrow \textit{onAlbum(i,a)}$$
$$\textit{getRelSongs}^{bff}(t_1, i_2, t_2) \leftarrow \textit{influenced}(i_1, i_2), \textit{hasTitle}(i_1, t_1), \textit{hasTitle}(i_2, t_2)$$

We have already seen that these functions lead to the enumeration of the domain of songs by unbound guessing plans. Now let us add the following function:

$$\textit{getSinger}^{bf}\textit{(t,s)} \leftarrow \textit{sang(s,i), hasTitle(i,t)}$$

This function retrieves the singer of a song. It has the same body as *getSongs*^{bff}, but a different binding pattern. We call it an *inverse function* of *getSongs*^{bff}, because it allows asking for the input parameter of *getSongs*^{bff}. If this function is provided, then the query that asks for the singer and album of *Hallelujah* can be answered by the following plan:

$$\textit{getSinger}^{bf}\textit{(Hallelujah,s), getSongs}^{bff}\textit{(s, Hallelujah, i),}$$
$$\textit{getAlbum}^{bf}\textit{(i,a)}$$

This plan first retrieves the singer of *Hallelujah*, s. Then, it retrieves all songs by s, and hopes that *Hallelujah* is one of these songs. Since the previous function call ensured that s is the singer of *Hallelujah*, this song will indeed be among the outputs of *getSongs*^{bff}. This call yields the song identifier i, which can then be used to call *getAlbum*^{bf} and retrieve the album. Thus, the addition of the inverse function allows us to produce answers to the query without guessing. We will now discuss how to prioritize such functions in execution plans.

B. Standard Approaches to Query Answering

TRANSFORMATION TO DATALOG. The standard way of answering queries with binding patterns is to transform the function definitions into inverse rules.[2] This yields a Datalog program, on which the query can be evaluated. An algorithm that is guaranteed to produce the maximal contained rewritings was introduced in [11]. Techniques for reducing the number of calls have been developed in [4, 5, 7]. Their goal remains the computation of the maximum number of answers. As we shall show next, these methods cannot prioritize plans with inverse functions over unbound plans.

INVERSE RULES. We illustrate the construction of the Datalog program proposed in [11] for our function.

$$\mathit{getRelSongs}^{bff}(t_1, i_2, t_2) \leftarrow \mathit{influenced}(i_1, i_2),\ \mathit{hasTitle}(i_1, t_1),\ \mathit{hasTitle}(i_2, t_2)$$

For each body atom, we construct an *inverse rule*. The atom forms the head of the inverse rule, while the body consists of a *dom* atom for every bound variable of the function, followed by the function call atom. In the example, this yields

$$\begin{array}{ll}
\mathit{influenced}(f_1, i_2) & \leftarrow \mathit{dom}(t_1),\ \mathit{getRelSongs}^{bff}(t_1, i_2, t_2)\\
\mathit{hasTitle}(i_2, t_2) & \leftarrow \mathit{dom}(t_1),\ \mathit{getRelSongs}^{bff}(t_1, i_2, t_2)\\
\mathit{hasTitle}(f_1, t_1) & \leftarrow \mathit{dom}(t_1),\ \mathit{getRelSongs}^{bff}(t_1, i_2, t_2)
\end{array}$$

where $f_1 = f(t_1, \mathit{getRelSongs})$ is a Skolem term [15] that replaces the existential variable i_1. Furthermore, we add one domain rule for every f-variable of the function. These rules look similar to the inverse rules, but have *dom* in their head:

$$\begin{array}{l}
\mathit{dom}(i_2) \leftarrow \mathit{dom}(t_1),\ \mathit{getRelSongs}^{bff}(t_1, i_2, t_2)\\
\mathit{dom}(t_2) \leftarrow \mathit{dom}(t_1),\ \mathit{getRelSongs}^{bff}(t_1, i_2, t_2)
\end{array}$$

In addition, we add a domain rule with an empty body for each constant of the query. For the query q_2, we get:

$$\mathit{dom}(\mathit{Hallelujah}) \leftarrow$$

This way, we can produce a Datalog program for a given query and a given set of function definitions. If the body predicates of a rule are evaluated left to right, then this program ensures that all input parameters of a function call are bound before the function is called.

EVALUATION STRATEGIES. Even if inverse functions are present, a Datalog evaluation strategy has to enumerate all unbound plans in the worst case. This is because these plans may be the only plans that yield results. Thus, unbound plans cannot be excluded upfront. In [11], the authors suggest to use a bottom-up approach for evaluating the new Datalog program. This is guaranteed to compute the maximum number of answers. However, it will also enumerate entire domains (e.g., all singers), and call all possible functions on all possible constants – no matter whether these calls contribute to the answer of the query or not. Obviously, this approach is infeasible in the context of Web services. Magic-set techniques [16] have been developed for optimizing bottom-up evaluations to ressemble top-down evaluations. This technique simulates the QSQ technique [15] for top-down evaluations. We consider next the top-down evaluation of the new Datalog program and we show that even if an inverse function can avoid a guessing plan, standard top-down evaluation techniques cannot prioritize such plans.

BLINDNESS TO INVERSE FUNCTIONS. Let us now consider the top-down evaluation of

$$q_2(a,s) \leftarrow \mathit{hasTitle}(i,\ \mathit{Hallelujah}), \mathit{onAlbum}(i,a), \mathit{sang}(s,i)$$

Figure 3 shows how the atom $hasTitle(i, Hallelujah)$ is expanded in two possible SLD (Selective Linear Definite) derivation trees using the rules of the new Datalog program. Although the SLD trees are constructed top-down, the evaluation of the calls is bottom-up.

The left derivation tree is the guessing strategy: It starts with *dom(Hallelujah)*, and then calls $\mathit{getRelSongs}^{bff}(t,i,\mathit{Hallelujah})$ with *t=Hallelujah*. It "hopes" that Hallelujah is a related song for itself. Then it calls $\mathit{getSongInfo}^{bfff}$ in a new branch. The right derivation tree, in contrast, uses the inverse function: It first calls $\mathit{getSinger}^{bf}(s,t)$ with *t=Hallelujah*. This yields the singer of *Hallelujah*, s. Then it calls $\mathit{getSongs}^{bff}(s,i,t)$ with that singer s. This yields the id of *Hallelujah*. A call to $\mathit{getAlbum}^{bf}(i,a)$ in a different branch will yield the album a.

However, nothing allows the Datalog processor to distinguish which of the two plans at Figure 3 is better. One may even think that the left plan is better since it uses less calls than the right one. This is because the crucial link between $\mathit{getSongs}^{bff}(s,i,t)$ and $\mathit{getSinger}^{bf}(s,t)$, namely the fact that both functions share the same body, gets lost in the transformation to Datalog. Worse, the left plan can be extended to an unbound guessing plan, in which the bottom *dom(Hallelujah)* is replaced by a branch where $\mathit{getRelSongs}^{bff}$ is repeatedly applied to its own outputs. Standard Datalog evaluation cannot prioritize inverse functions over guessing plans, because the link between $\mathit{getSongs}^{bff}(s,i,t)$ and its inverse is lost.

C. Our Approach to Query Answering

SMART FUNCTION CALLS. We will now make the notion of "guessing" formal. We aim to distinguish the guessing plan

$$\mathit{getRelSongs}^{bff}(\mathit{Hallelujah}, i_2, t_2),\ \mathit{getSongInfo}^{bfff}(i_2, \mathit{Hallelujah}, s, a)$$

from the "smart plan"

$$\mathit{getSinger}^{bf}(\mathit{Hallelujah}, s),\ \mathit{getSongs}^{bff}(s,\ \mathit{Hallelujah},\ i),\ \mathit{getAlbum}^{bf}(i,a)$$

Definition 6 (Smart Function Call): A smart function call in an execution plan is a function call whose inputs come from the query or from previous smart function calls, and whose consequences are a subset of the consequences of the following function calls.

In the example, the call to $\mathit{getSinger}^{bf}(\mathit{Hallelujah},\ s)$ is a smart function call, because its consequences ($\{\mathit{sang}(s,\mathit{Hallelujah})\}$) are a subset of the consequences of the following calls, $\{\mathit{sang}(s,\mathit{Hallelujah}), \mathit{hasTitle}(i,\mathit{Hallelujah}), \mathit{onAlbum}(i,a)\}$. The call $\mathit{getRelSongs}^{bff}$ of the left plan, in contrast, is not a smart

[2] Inverse rules have nothing to do with the inverse *functions* of SUSIE.

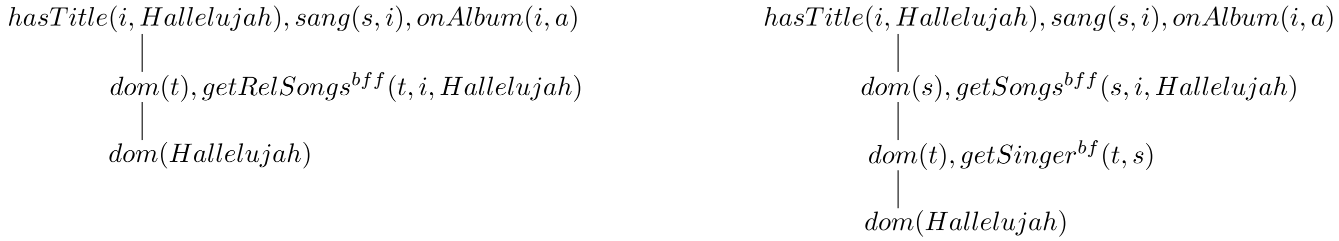
$hasTitle(i, Hallelujah), sang(s, i), onAlbum(i, a)$
$dom(t), getRelSongs^{bff}(t, i, Hallelujah)$
$dom(Hallelujah)$
$hasTitle(i, Hallelujah), sang(s, i), onAlbum(i, a)$
$dom(s), getSongs^{bff}(s, i, Hallelujah)$
$dom(t), getSinger^{bf}(t, s)$
$dom(Hallelujah)$

Fig. 3. Sample SLD derivations for q: without the inverse function (left), and with the inverse function (right).

function call. It guesses that its output will be the desired input of the following call to *getSongInfo*^{bfff}. Obviously, smart function calls should be preferred wherever possible. We can give the following theorem to support this intuition.

Theorem 1 (Smart Function Calls): Given a query Q, and given an effective execution plan E for Q, such that there is some choice of input variables such that E returns answers to Q, the following holds: If we can add smart function calls to E, so that E becomes admissible, then E will deliver an answer for Q.

Proof 1: Assume that there is an effective execution plan E for a query Q, such that there is some choice of input variables such that E returns an answer A for Q. Consider a function call $f(\bar{X})$, so that the consequences of $f(\bar{X})$ are a subset of the consequences of E. Be $f \wedge E$ the conjunction of the consequences of $f(\bar{X})$ and the consequences of E. Then the evaluation of $f \wedge E$ on the global schema has the same results as the evaluation of the consequences of E on the global schema. Hence, the evaluation of $f \wedge E$ will contain A. If $f(\bar{X})$ makes E admissible, then E with $f(\bar{X})$ will deliver A.

Intuitively, Theorem 1 tells us that smart function calls will not deteriorate the chance of obtaining an answer from a function composition. That is, if we have a function composition that can deliver answers to the query, but where we need to bind the inputs, then adding the smart function calls to bind the inputs is a safe step. Note that such a proof cannot be given for non-smart function calls. This justifies the preference of smart function calls over non-smart function calls.

ALGORITHM. We will now discuss how to prioritize smart function calls in SLD derivations. The key idea is that every function call has to answer either an unanswered query atom, or to have consequences that are contained in the conjunction of the consequences of the following calls. In the right plan in Figure 3, the consequences of *getSinger*^{bf} are:

sang(s,i), hasTitle(i, t)

These consequences are contained the consequences of the following calls *getSongs*^{bff} and *getAlbum*^{bf}:

sang(s, i), hasTitle(i, Hallelujah), onAlbum(i,a)

Therefore, this call is a smart function call and the plan should be prioritized. Let us now look at the left SLD tree. The consequences of *getRelSongs*^{bff} are:

hasTitle(i, Hallelujah), influenced(i", i), hasTitle(i", t)

These consequences are not in the consequences of the following call:

hasTitle(i, Hallelujah), sang(s,i), onAlbum(i,a)

Therefore, this plan should be postponed. Non-smart function calls cannot always be avoided. But plans with less non-smart calls are more likely to succeed. This ordering strategy can be added to a standard Datalog processor. This allows the query evaluation to prioritize smart function calls over guessing function calls.

V. INVERSE FUNCTIONS

The previous section has shown how inverse functions can be prioritized in execution plans. The present section will show how inverse functions can be generated and implemented.

A. Definition

ASYMMETRIC RELATIONS. Consider a function definition that contains a relation r:

$$f^{b\dots bf\dots f}(x_1, \dots, x_i, x_{i+1}, \dots, x_n) \leftarrow \dots r(x_k, \dots, x_l) \dots$$

Let us consider a query atom $r(y_k, \dots, y_l)$, in which some arguments are bound and some arguments are unbound. Thus, the query atom defines a binding pattern of bound and free variables on r, $\bar{A}_r$. If $\bar{A}_r$ does not provide bound values for all input arguments of f, then f cannot be called. We say that f *does not support* the binding pattern $\bar{A}_r$. If the set of functions F contains no function that supports $\bar{A}_r$, then we say that $\bar{A}_r$ is *unsupported* in F. If there exists a binding pattern of r that is supported in F, and if there exists another binding pattern that binds at least one variable and that is unsupported in F, then we say that r is *asymmetric* with respect to F. Intuitively speaking, this means that there are arguments of r for which we can query and there are other arguments for which we cannot query.

INVERSE FUNCTIONS. If a function f does not support a binding pattern of a relationship, then the query evaluation might have to enumerate an entire domain to "guess" input values. Therefore, we introduce the *inverse functions* of f.

Definition 7 (Inverse Function): Given a function $f^{\bar{A}}(\bar{X}_0)$, an inverse function of f is a function $f^{\bar{A}'}(\bar{X}_0)$ with the same rule body. $\bar{A}'$ is an adornment that has at least one bound argument, and that leaves at least one input argument of f free. As an example, consider again the following function:

getSongs^{bff}*(s,i,t)* $\leftarrow$*sang(s,i), hasTitle(i,t)*

This function requires the singer as input, and delivers the song id and title as output. It has 3 inverse functions:

getSongs^{ffb}*(s,i,t), getSongs*^{fbf}*(s,i,t), getSongs*^{fbb}*(s,i,t)*

Adding Inverse Functions. Inverse functions can be used to answer query atoms that have an otherwise unsupported binding pattern. Yet, inverse functions are not always necessary. If, say, there is a function f that supports the desired binding pattern for a relation, then it is not necessary to add an inverse function for some other function g that does not support the desired binding pattern. However, in the context of Web services, the inverse of g would still be necessary in order to obtain as many results as possible. This is because Web services are typically incomplete. The function f may not yield all the instances that g stores. Thus, one potentially loses results if one queries only f.

Even if f is complete, it can still be beneficial to have the inverse of g. This is because Web services typically come at a cost. This can be a cost in bandwidth, in time, or also in financial terms (for commercial Web services). It can be that the inverse of g is cheaper than f. In this case, the inverse of g can still be preferable to f.

Now assume that there is an orchestration of functions that allows finding the input values for g, so that g delivers instances even for an unsupported binding pattern. Since Web services tend to be incomplete, this orchestration might still not find *all* input values of g. Thus, g might store more instances than can be obtained this way. Thus, the inverse of g is still necessary to maximize the number of results.

In all of these cases, the inverse of g proves useful to have – be it to maximize the number of results or to minimize the cost of calls. Therefore, we aim to add as many inverse functions to our program as possible.

B. Implementation

Providing Inverse Functions. We will now explain how certain inverse functions can be provided and implemented. Our method is intended for cases where at most one input argument of the original function is unbound, and where the necessary information appears on the surface Web. We discuss these limitations in Section V-C. As an example, consider again the function $getSongs$:

getSongs^{bff}*(s,i,t)* $\leftarrow$*sang(s,i), hasTitle(i,t)*

Information about singers and songs are likely to appear on the Web. Therefore, we can provide the inverse functions where one of these values is bound. As an example, consider

getSongs^{ffb}*(s,i,t)* $\leftarrow$*sang(s,i), hasTitle(i,t)*

Our system, SUSIE, implements this inverse function as follows. Whenever *getSongs*^{ffb}*(s,i,t)* is called, SUSIE issues a keyword query of the form "Singer t" to a Web search engine (e.g., "Singer Hallelujah"). Then, SUSIE will extract possible candidates for s from the result Web pages. Last, SUSIE will call the real Web service *getSongs*^{bff}*(s,i,t)* with all candidates for s. If any of them succeeds and delivers the given t as output, then SUSIE returns the values of s and i as an output of the inverse function *getSongs*^{ffb}. Thereby, the inverse function acts just like a real Web service function. We will now discuss three subtasks of this endeavor: (1) the task of finding suitable Web pages, (2) the task of extracting candidates from the Web pages, and (3) the task of verifying the results. In all of the following, we treat the generic case of implementing the inverse functions for a function $f^{b\ldots bf\ldots f}(x_1, \ldots, x_n)$, where the first argument of the inverse function is unbound.

Finding Web Pages. We assume that the function f comes with a domain for its free argument x_1, the *target type*. In the example of *getSongs*, we know that the target type is *Singer*. We map each inverse function $f^{\bar{A}}$ of f to a *keyword query*. This keyword query is a string of the form $tt\ \bar{X}$, where tt is the target type and $\bar{X}$ are the bound variables of $\bar{A}$. In the example, the keyword queries are

getSongs^{ffb}*(s,i,t)*: "singer t"
getSongs^{fbb}*(s,i,t)*: "singer t i"

In actual implementations, the keyword queries can be more sophisticated. For example, it may turn out to be beneficial to map *getSongs*^{fbb}*(s,i,t)* to the keyword query "List of singers who sang t", ignoring the song id i. Whenever the query evaluation calls the function $f^{\bar{A}}(x_1, \ldots, x_n)$, SUSIE issues the keyword query of $f^{\bar{A}}$ to an Internet search engine (we used Google). SUSIE collects the top ten result Web pages.

Extracting Candidates. Once the Web pages have been retrieved, it remains to extract the candidate entities. Information extraction is a challenging endeavor, because it often requires near-human understanding of the input documents. Our scenario is somewhat simpler, because we are only interested in extracting the entities of a certain type from a set of Web pages. We have implemented two simple yet effective IE algorithms as a proof of concept.

String Matching Algorithm. This algorithm extracts only entities that are already known to a knowledge base. In our experiments, we use the YAGO knowledge base [1]. YAGO feeds from Wikipedia and thus covers a large number of entities of common interest. The algorithm first loads all entities of the target type from the knowledge base into a trie [17]. Then the algorithm runs through the Web pages and extracts all entities from the documents that appear in the trie. This processing can be done in time $O(n)$ in the best case and in time $O(m \cdot n)$ in the worst case, where n is the total number of characters in the Web pages and m is the number of characters in the longest entity name.

Structured Extraction Algorithm. The String Matching Algorithm has the disadvantage that it can only find entities that appear in the knowledge base. If we wish to venture beyond this limitation and find new entities, we may exploit that many result Web pages will have a structured form. Typically, tables represent a natural way to organize sets of relationships in Web pages. However, as shown in [18], only a small fraction of the Web tables are encoded using the `<table>` markup in HTML. In many cases, they are encoded using lists or loosely repetitive structures. Our algorithm identifies structures of repetitive rows, where each row contains items that are separated by special strings or tags that re-appear in each row. Furthermore, the items in one column have to be of the same syntactic type (numbers, strings or dates). This procedure finds standard tables and standard lists as well as other types of repetitive structures. By comparing the elements of each column with the instances of the target type in YAGO, our algorithm finds the column that constitutes most likely the answers to the query.

We note that these are just two possible implementations. They can be replaced by more sophisticated ones [19, 20].

VERIFYING CANDIDATES. The IE algorithms have delivered a set of candidate entities for the free argument x_1 of f. In case of the String Matching Algorithm, these candidates have been filtered by a knowledge base. Still, this does not mean that the candidates would be correct: Web pages can contain many more entities of the target type than the desired ones. Therefore, all candidates have to be checked by the Web service to see whether they fulfill the conditions of the function definition. To do so, SUSIE will call the original Web service function f with all of these candidate entities, one after the other. This yields one, multiple, or no result tuples of the form $\langle x_1, \ldots, x_n \rangle$ for each candidate entity x_1. If the values in the bound positions of $\bar{A}$ correspond to the input values of the inverse function call $f^{\bar{A}}(x_1, \ldots, x_n)$, then SUSIE delivers the tuple $\langle x_1, \ldots, x_n \rangle$ as an output of the inverse function. Thereby, each candidate entity that has been extracted from the Web pages is verified by the Web service.

C. Properties of SUSIE

IE AND WEB SERVICES. The candidate entities often appear multiple times in the Web pages. This increases the chances of an IE algorithm to find them. A high precision and a high recall are desirable for the IE algorithms, but they are not strictly necessary. If the precision of the extraction is low, this will result in more Web service calls, but not in diminished precision of the final query answers. This is because all answers are checked by the Web service. If the recall this low, this will result in fewer answers. Fewer answers, however, are better than trying out all possible input values for the function, which may result in no answer at all due to the limited call budget. By verifying each result with the original Web service, our approach mitigates the central weakness of classical information extraction, its imperfect precision. By using information extraction to extract candidates, our approach mitigates the central limitation of Web services, the restrictive access patterns.

LIMITATIONS OF SUSIE. The current implementation of SUSIE creates inverse functions only if at most one input argument of the original function is unbound. The case of functions with multiple unbound inputs is significantly harder: It could require fact extraction of n-ary facts. This is, by itself, a hard problem. If one uses a naive entity recognition algorithm (such as the string matching algorithm), then one can generate a number of candidate tuples that is exponential in the number of inputs. Therefore, we leave the case of multiple inputs for future work. As we show in the experiments, even the case with one input delivers significant mileage in practice.

We can implement inverse functions only if good candidate entities for a query can be found on the Web. This is certainly not true for a large number of functions. In these cases, we do not provide the inverse functions. However, experience from our experiments indicates that information of common interest is publicly available on the Web in a large spectrum of cases. In these cases, we can provide the inverse functions. Our claim is not that SUSIE could make all queries answerable. Rather, our claim is that SUSIE can make queries answerable in an interesting spectrum of cases. Our experiments show that in these cases, SUSIE improves the query results drastically.

VI. PERFORMANCE EVALUATION

We conducted 2 types of experiments. We first evaluate the performance of the information extraction algorithms. Then, we evaluate the performance of SUSIE on real-world queries.

A. Information Extraction

TEST SET. To evaluate the IE algorithms, we targeted three query types: Queries that ask for actors with a certain birth year, for actors with a certain nationality and for authors who received a certain prize. For each query type, we chose 10 arbitrary property values (10 birth years, 10 nationalities and 10 literature prizes). For each property value, we generated the keyword query that SUSIE would generate, sent it to Google and retrieved the top 10 pages. This gave us 100 pages for each test set. The pages are quite heterogeneous, containing lists, tables, repetitive structures and full-text listings of entities. We manually extracted the target entities from these pages to create a gold standard. Then, we ran the IE algorithms and measured their performance with respect to the gold standard.

DATABASE COMPETITOR. In SUSIE, the extraction algorithms are used to generate candidates for the input values for Web services. Query evaluation algorithms without this capability need to generate the input values for Web services by enumerating a domain. For example, while SUSIE will guess actors born in 1970 by searching for “actors 1970” on the Web, a non-IE-based query evaluation will guess actors born in 1970 by enumerating all available actors. To judge how many of the candidate actors would actually be actors born in 1970, we report the number of entities of the target type in the YAGO database that have the desired property. For example, for actors born in 1970, we report the proportion of actors in YAGO that are born in 1970 (among those actors that have a birth date). This is an estimator for the chance that a candidate generated by enumerating a domain will be a valid input value for the Web service.

RESULTS. Figures 4, 5, and 6 and show the results. Every row contains the values averaged for 10 Web pages. #E is the average number of entities per page. SMA is the String Matching Algorithm, and SEA is the Structured Extraction algorithm. The column “DB” is the naive algorithm of regarding all instances of the target type in the YAGO database as candidates. The precision and recall of the IE algorithms are nearly always in the range between 30% and 75%. Only the precision on the birth year queries is disappointing, with values below 10% (Figure 5). This is because the Google queries returned lists of all actors, not just of the ones born in a certain year. Thus, the algorithms find far too many irrelevant entities in the pages. The SMA, with its slightly higher recall, suffers particularly for the precision. We record this as a case where the information extraction approach is less practical, because the Internet does not provide the lists of entities that the approach needs.

Award	#E	SMA		SEA		DB
		Prec	Rec	Prec	Rec	Prec
Franz Kafka	2	25 %	73 %	13 %	34 %	N/A
Golden Pen	9	36 %	33 %	29 %	56 %	N/A
Jerusalem	6	23 %	52 %	69 %	24 %	N/A
National Book	69	38 %	59 %	45 %	76 %	0.9 %
Nobel Prize	44	41 %	29 %	46 %	40 %	2.9 %
Phoenix	4	47 %	71 %	18 %	76 %	N/A
Prix Decembre	4	29 %	6 %	18 %	25 %	N/A
Prix Femina	21	31 %	13 %	32 %	32 %	0.6 %
Prix Goncourt	73	63 %	46 %	7 %	1 %	1.12%
Pulitzer	42	78 %	79 %	60 %	46 %	2.0 %
	27	43 %	44 %	34 %	35 %	1.5%

Fig. 4. IE Results for “Authors who won prize X”

Year	#E	SMA		SEA		DB
		Prec	Rec	Prec	Rec	Prec
1940	2	2 %	73 %	1 %	80 %	0.8 %
1945	1	2 %	96 %	1 %	100 %	1.0 %
1950	2	2 %	81 %	1 %	83 %	1.2 %
1955	2	6 %	39 %	3 %	56 %	1.2 %
1960	18	12 %	60 %	6 %	72 %	1.3 %
1965	8	17 %	72 %	14 %	71 %	1.5 %
1970	4	20 %	96 %	1 %	66 %	1.7 %
1975	2	8 %	91 %	1 %	67 %	1.6 %
1980	6	8 %	52 %	4 %	90 %	1.6 %
1985	2	8 %	56 %	0 %	43 %	1 %
	5	9 %	71 %	3 %	74 %	1.3 %

Fig. 5. IE Results for “Actors born in year X”

Country	#E	SMA		SEA		DB
		Prec	Rec	Prec	Rec	Prec
Australia	15	37 %	82 %	51 %	66 %	11%
Canada	5	28 %	92 %	40 %	50 %	20%
England	46	46 %	85 %	71 %	74 %	0 %
France	153	48 %	42 %	50 %	64 %	2 %
Germany	45	50 %	57 %	51 %	99 %	2 %
Greece	26	38 %	58 %	2 %	14 %	0 %
Italy	138	29 %	54 %	42 %	59 %	0 %
Mexico	25	44 %	52 %	51 %	78 %	0 %
South Africa	12	29 %	76 %	29 %	63 %	0%
Spain	24	54 %	63 %	67 %	94 %	0 %
	47	38 %	65 %	46 %	63 %	3.5 %

Fig. 6. IE Results for “Actors of nationality X”

DISCUSSION. A precision of 30% may not sound extraordinary. Yet, it has to be seen in comparison to the naive approach of sending all entities of the target type to the Web service. In general, the proportion of entities in the database that have the desired property is very low. The percentages for writer awards are already an overestimation, because they consider only those writers that did win an award, while many writers do not win any award at all in their life. So let us e.g. assume that 1% of the entities have the desired property. This means that an expected 100 calls would have to be sent to the Web service before finding one of them. This number of calls is already above the budget we are considering, meaning that the user would likely not get any response at all. An IE precision of 30%, in contrast, means that for every 3 queries that are sent to the Web service, only 2 are sent in vain. Likewise, a recall of 30% means that we can find one third of the entities that the user is potentially interested in – as opposed to none if the call budget is exhausted by enumerating a domain. Thus, even in the cases with lower precision, our approach allows answering queries that would be impossible to answer otherwise.

Service	Function
Music-Brainz	$getArtists_{mb}^{bfff}(artist, id, born, died)$ $getAlbums_{mb}^{bfff}(album, id, artist, releaseDate)$
Abe-Books	$getBooksByTitle_{abe}^{bffff}(title, id, isbn, author, publisher)$ $getBooksByIsbn_{abe}^{bffff}(isbn, title, id, author, publisher)$ $getBooksByAuthor_{abe}^{bffff}(author, title, id, isbn, publisher)$
Library-Thing	$getAuthors_{lt}^{bfffff}(x, born, prize, country, school, place)$ $getBooks_{lt}^{bffffff}(title, author, prize, publicationDate)$

Fig. 7. Some of the functions integrated in SUSIE

B. Real-world Queries

In this section, we evaluate our approach on real queries with real Web Services.

WEB SERVICES AND QUERIES. We integrated 40 functions exported by 7 Web service providers: *isbndb.org*, *librarything.com*, *Amazon*, *AbeBooks*, *api.internetvideoarchive.com*, *musicbrainz.org*, *lastfm.com*. Figure 7 shows the signatures of some of these functions. We selected a variety of query templates, which can be organized in the following classes (Figure 8): star queries with constants at the endpoints (Q_1-Q_2, Q_7), star queries with variables and constants at the endpoints (Q_3-Q_4, Q_8-Q_{10}), and chain queries with constants at the endpoints (Q_5-Q_6, Q_{11}). For every query template, we evaluate a set of similar queries by varying the constants. The queries were chosen such that they have different alternative ways of composing function instantiations. Usually, this leads to a high number of Web service calls.

SETTINGS AND ALGORITHMS. We distinguish two settings. In the first setting we try to answer the query using only Web services. We compare 3 different approaches. The first approach (“TD”) uses a naive top-down evaluation of the queries without inverse functions. This approach implements a Prolog-style backtracking strategy. The second approach uses ANGIE [12] for the query evaluation. ANGIE is a state-of-the-art system for top-down query evaluation with views with binding patterns. The third approach uses SUSIE, i.e., the approach uses both Web services and inverse functions. We used the SEA algorithm for IE.

In the second setting, we allow the approaches to make use of the YAGO knowledge base [1]. The first approach in this setting is a baseline solution, which simply issues the query to YAGO, and does not call any functions. The second approach is YAGO+TD, which is allowed to use both functions and the YAGO knowledge base. This means that it can retrieve answers from YAGO, it can retrieve answers from functions, and it can also use data from YAGO to call the functions. The third approach is YAGO+ANGIE, and the fourth is YAGO+SUSIE. Only SUSIE uses inverse functions. For all the algorithms, we set the budget to 15 for the number of calls to one service and to 100 for the total number of calls. As performance metrics, we measured the total number of answers output by each algorithm.

RESULTS. Figure 9 shows the results for the queries

No.	Query		Constants
Q1	type	(?person, Writer)	p
	wonAward	(?person, p)	
Q2	type	(?person, Writer)	p, c
	wonAward	(?person, p)	
	isCitizenOf	(?person, c)	
Q3	type	(?person, Writer)	p
	wonAward	(?person, p)	
	wrote	(?person, ?book)	
Q4	type	(?person, Writer)	p, c
	wonAward	(?person, p)	
	isCitizenOf	(?person, c)	
	wrote	(?person, ?book)	
Q5	type	(?person, Writer)	p, y
	wonAward	(?person, ?prize)	
	isTitled	(?prize, p)	
	awardedInYear	(?prize, y)	
Q6	type	(?person, Writer)	p
	wrote	(?person, ?book)	
	wonAward	(?person, ?prize)	
	isTitled	(?prize, p)	
	awardedInYear	(?prize, y)	
Q7	type	(?person, Actor)	c
	isFamousActor	(?person, True)	
	isCitizenOf	(?person, c)	
Q8	type	(?person, Actor)	c
	isFamousActor	(?person, True)	
	isCitizenOf	(?person, c)	
	actedIn	(?person, ?movie)	
Q9	type	(?person, Actor)	p
	wonAward	(?movie, p)	
	actedIn	(?person, ?movie)	
Q10	type	(?person, Actor)	p
	wonAward	(?movie, p)	
	producedIn	(?movie, ?country)	
	actedIn	(?person, ?movie)	
Q11	type	(?person, Singer)	p, y
	sang	(?person, ?song)	
	wonAward	(?person, ?prize)	
	isTitled	(?prize, p)	
	awardedInYear	(?prize, y)	

Fig. 8. Query templates

corresponding to the templates in Figure 8. We report the number of answers for all algorithms in both settings: With only functions and with both functions and YAGO. In both settings, all algorithms consume their entire budget of calls. Since all algorithms use the same number of calls, the total number of answers returned by each serves as comparison metric. For SUSIE we show the number of calls that were consumed by the algorithm for two moments in time: when the first answer was output (#c1a) and when the final answer was output (#cFa). Subsequent calls used up the budget, but did not return answers.

In the first setting, where only functions can be used, only SUSIE delivers any answers at all. This is because algorithms without inverse functions have to compose existing functions to compute answers, which often consumes the entire budget before any answer is returned. It may also be just impossible to find such a composition. In the second setting, the algorithms can also use YAGO. YAGO already contains some answers to the queries (reported in the column "YAGO"). All algorithms first return these answers from YAGO, and then embark to call functions. We observe that, for all queries, SUSIE returns more answers than the two other algorithms, or at least an equal number. For instance, for the first query of the template Q_4, SUSIE outputs almost twice as many answers as ANGIE.

If we compare SUSIE with YAGO to SUSIE using just functions, we remark that using YAGO improves the number of answers in most cases. In some cases SUSIE performs better without YAGO. This is because the query planner sometimes chooses to enumerate a domain from YAGO, even if the inverse function is present. In almost all cases, SUSIE improves over the answers that YAGO alone provides. Even for queries that have no answers in YAGO ($Q3$ - four, $Q6$ - four, $Q9$, $Q10$, $Q11$), SUSIE makes smart use of the database to double the number of answers with respect to the case where just functions are used.

VII. Related Work

Query Answering. Most related to our setting is the problem of answering queries using views with limited access patterns [3]. The approach of [3] rewrites the initial query into a set of queries to be executed over the given views. The authors show that for a conjunctive query over a global schema and a set of views over the same schema, determining whether there exists a conjunctive query plan over the views that is equivalent to the original query is NP-hard in the size of the query. This rewriting strategy assumes that the views are complete (i.e., contain all the tuples in their definition). This assumption is unrealistic in our setting with Web services, where sources may overlap or complement each other but are usually incomplete.

When sources are incomplete, one aims to find maximal contained rewritings of the initial query, in order to provide the maximal number of answers. [11] present algorithms for rewriting a query into a Datalog program, requiring recursive Datalog even if the initial query is non-recursive. Subsequent studies [4–7, 21, 22] proposed solutions for reducing the number of accesses. Notions of minimal rewritings have been proposed in [7, 8]. However, the goal remains the computation of maximal results. The guessing accesses are not eliminated nor do they have a special treatment since they relevant for this goal. The same problem was studied for different query languages: unions of conjunctive queries with negation [9], with additional function dependencies [10], or with integrity constraints [8]. The Magic Set algorithm [16] reduces the number of sub-queries in a bottom-up evaluation.

Our own ANGIE system [12] also answers queries using views as surrogates for Web services, and it imposes an upper bound on the number of function calls. The strategy is to prioritize function calls that are more likely to deliver answers.

None of these approaches can cope with situations where the available functions have binding patterns that do not allow answering the query. For example, if there is only one function, $getSongs^{bf}(singer, song)$, and the user asks who sang "Hallelujah", then no query evaluation procedure, no optimization strategy, and no smart orchestration mechanism can deliver an answer to the query. This is because the function cannot be called without a singer. This limitation applies to all approaches listed above. What is needed in such cases are the inverse functions that SUSIE provides.

Deep Web Querying. A Deep Web page is a form, which requires certain values to be filled in, and which delivers results for these values. Thus, guessing the right

Q	Constants	Just Functions					YAGO	YAGO	YAGO	**YAGO**		
	YAGO	TD	ANGIE	**SUSIE**	#c1a	#cFa		+TD	+ANGIE	**+SUSIE**	#c1a	#cFa
Q1	Nobel Prize in Literature	0	0	**14**	3	55	103	103	103	**103**	0	0
	Golden Pen Award	0	0	**11**	4	16	0	0	0	**11**	4	16
	Franz Kafka Prize	0	0	**5**	4	8	0	0	0	**5**	4	8
	American Book Medal	0	0	**16**	3	18	0	0	0	**16**	3	18
	Jerusalem Prize	0	0	**11**	3	21	0	0	0	**11**	3	21
Q2	France, Nobel Prize Literature	0	0	**5**	2	9	6	6	6	**9**	0	8
	UK, Franz Kafka Prize	0	0	**1**	2	2	0	0	0	**1**	2	2
Q3	Nobel Prize Literature	0	0	**198**	43	100	234	235	453	**457**	0	94
	Golden Pen Award	0	0	**228**	18	87	0	0	0	**226**	6	99
	Franz Kafka Prize	0	0	**132**	19	97	0	0	0	**181**	5	92
	American Book Medal	0	0	**296**	19	97	0	0	0	**522**	3	111
	Jerusalem Prize	0	0	**220**	22	90	0	0	0	**233**	4	91
Q4	France, Nobel Prize Literature	0	0	**144**	11	89	2	61	74	**133**	0	107
	UK, Franz Kafka Prize	0	0	**79**	3	63	0	0	0	**70**	3	61
Q5	Nobel Prize Literature 2004	0	0	**1**	2	2	0	0	0	**1**	2	2
	Golden Pen Award 2006	0	0	**1**	2	2	0	0	0	**1**	2	2
	Franz Kafka Prize 2006	0	0	**1**	2	2	0	0	0	**1**	2	2
	American Book Medal	0	0	**1**	2	2	0	0	0	**1**	2	2
	Jerusalem Prize 1981	0	0	**1**	2	2	0	0	0	**1**	2	2
Q6	Nobel Prize in Literature 2004	0	0	**31**	3	51	0	0	0	**31**	3	51
	Golden Pen Award 2006	0	0	**57**	3	52	0	0	0	**64**	3	51
	Franz Kafka Prize 2006	0	0	**61**	3	59	0	0	0	**89**	2	59
	American Book Medal	0	0	**77**	3	75	0	0	0	**243**	2	85
	Jerusalem Prize 1981	0	0	**60**	3	71	0	0	0	**90**	2	69
Q7	United States Of America	0	0	**7**	5	20	0	0	0	**7**	5	20
	United Kingdom	0	0	**7**	3	26	0	0	0	**7**	3	26
Q8	United States Of America	0	0	**309**	5	40	0	0	0	**330**	5	40
	United Kingdom	0	0	**213**	3	52	0	0	0	**234**	3	66
Q9	Academy Award Best Picture	0	0	**85**	2	56	0	0	0	**187**	2	88
Q10	Academy Award Best Picture	0	0	**79**	2	16	0	0	0	**212**	2	94
Q11	Grammy Awards 2009	0	0	**110**	69	75	0	0	0	**377**	69	75

Fig. 9. Number of answers for the queries of the templates in Figure 8

values for the forms has similarities to guessing the right input values for Web services. Much work has addressed the probing, semantic categorization, or materialization of Deep Web forms. Google's "surfacing" technique [23] aims to materialize the Deep Web by determining the most promising input values for the forms. [24] aims to match the schema of two Deep Web forms. This approach finds instances for Deep Web attributes by querying the surface Web, and then validating the instances through the original Deep Web form. [25] estimates the domain and the size of the domain of Deep Web attributes by systematic probing. Other work [26, 27] has focused on probing form fields and reconstructing the "schema" of a single form, so that queries can place properly typed values into specific fields, avoiding unnecessary requests.

These works differ in three aspects from our setting. First, the approaches typically analyze the Deep Web form in an off-line fashion, where all necessary information is available before query time. This allows for approaches that use training and learning. In our setting, in contrast, information has to be extracted and integrated on the fly, because it depends dynamically on the constants of the query. This restricts the set of applicable methods to techniques that work at query time. Second, work on the Deep Web typically aims to fill unary relations (e.g., finding all reasonable values for the field "author"). The population of unary relations can draw upon a large pool of techniques (such as Hearst patterns [28]). In our setting, in contrast, we have to find arguments for binary relations (e.g., the "author of the book 'Don Quixote' "). Thus, the techniques from the Deep Web do not directly transfer to our setting. Third, our setting requires the dynamic composition of Web service functions in order to answer user queries. The IE-based functions have to be integrated into this orchestration and called on the fly. The above work on the Deep Web does not address the prioritization of inverse functions in execution plans.

INFORMATION EXTRACTION. Information extraction (IE) is concerned with extracting structured data from documents. IE methods suffer from the inherent imprecision of the extraction process. Usually, the extracted data is way too noisy to allow direct querying. SUSIE overcomes this limitation, by using IE solely for finding candidate entities of interest and feeding these as inputs into Web service calls. Named Entity Recognition (NER) approaches [29–31] aim to detect interesting entities in text documents. They can be used to generate candidates for SUSIE. The first approach discussed in this paper matches noun phrases against the names of entities that are registered in a knowledge base – a simple but effective technique that circumvents the noise in learning-based NER techniques. The second IE approach used in this paper is to extract structured data from Web tables [18–20]. For SUSIE, we have developed judiciously customized methods along these lines. These are not limited to lists and tables, but detect arbitrary repetitive structures that could contain candidates. Alternative IE methods such as Wrapper Induction [32], fact extraction [33–35], or entity extraction [31, 36] could be also considered, but they are not practical in our scenario as they require training data and, thus, human supervision.

SCHEMA AND ENTITY MATCHING. In this paper, we assume that all Web services have been mapped to the same schema. There is ample literature on the (semi-) automatic creation of schema mappings [37]. Likewise, we are not concerned with entity disambiguation and data fusion [38, 39]. Both tasks are required for SUSIE, but orthogonal to our contribution on inverse functions and smart execution plans.

WEB SERVICE ORCHESTRATION. There is plenty of prior work on Web service orchestration. The goal is to describe complex business processes that are carried out using Web service compositions. The standard for specifying such workflows is BPEL [40]. A suite of papers addressed problems ranging from verification issues [41] to optimization [42]. The works of [43–46] devise an architecture for orchestration of several sources. However, none of these works addresses the issue of service asymmetry.

VIII. CONCLUSION

This paper has introduced the problem of asymmetric Web services. We have shown that a considerable number of real-world Web services allow asking for only one argument of a relationship, but not for the other. We have proposed to use information extraction to guess bindings for the input variables and then validate these bindings by the Web service. Through this approach, a whole new class of queries has become tractable. We have shown that providing inverse functions alone is not enough. They also have to be prioritized accordingly. We have implemented our system, SUSIE, and showed the validity of our approach on real data sets. We believe that the beauty of our approach lies in the fruitful symbiosis of information extraction and Web services, which each mitigate the weaknesses of the other.

Our current implementation uses naive information extraction algorithms that serve mainly as a proof of concept. Future work will explore new algorithms that could step in. We also aim to automize the discovery of new Web services and their integration into the system.

REFERENCES

[1] F. M. Suchanek, G. Kasneci, and G. Weikum, “YAGO: A Core of Semantic Knowledge,” in *WWW*, 2007.

[2] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, “DBpedia: A nucleus for a Web of Open Data,” *Semantic Web*, 2008.

[3] A. Rajaraman, Y. Sagiv, and J. D. Ullman, “Answering queries using templates with binding patterns,” in *PODS*, 1995.

[4] C. Li and E. Y. Chang, “Query planning with limited source capabilities,” in *ICDE*, 2000.

[5] C. Li, “Computing complete answers to queries in the presence of limited access patterns,” *VLDB J.*, 2003.

[6] S. Kambhampati, E. Lambrecht, U. Nambiar, Z. Nie, and G. Senthil, “Optimizing recursive information gathering plans in EMERAC,” *J. Intell. Inf. Syst.*, 2004.

[7] A. Calì and D. Martinenghi, “Querying data under access limitations,” in *ICDE*, 2008.

[8] A. Deutsch, B. Ludäscher, and A. Nash, “Rewriting queries using views with access patterns under integrity constraints,” *Theor. Comput. Sci.*, 2007.

[9] A. Nash and B. Ludäscher, “Processing unions of conjunctive queries with negation under limited access patterns,” in *EDBT*, 2004.

[10] A. Calì, D. Calvanese, and D. Martinenghi, “Dynamic query optimization under access limitations and dependencies,” *J. UCS*, 2009.

[11] O. M. Duschka, M. R. Genesereth, and A. Y. Levy, “Recursive query plans for data integration,” *J. Log. Program.*, 2000.

[12] N. Preda, G. Kasneci, F. M. Suchanek, T. Neumann, W. Yuan, and G. Weikum, “Active Knowledge : Dynamically Enriching RDF Knowledge Bases by Web Services. (ANGIE),” in *SIGMOD*, 2010.

[13] R. Fagin, L. M. Haas, M. A. Hernández, R. J. Miller, L. Popa, and Y. Velegrakis, “Clio: Schema mapping creation and data exchange,” in *Conceptual Modeling: Foundations and Applications*, 2009.

[14] C. T. Kwok and D. S. Weld, “Planning to gather information,” in *AAAI/IAAI, Vol. 1*, 1996.

[15] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.

[16] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman, “Magic sets and other strange ways to implement logic programs,” in *PODS*, 1986.

[17] E. Fredkin, “Trie memory,” *Commun. ACM*, no. 9, September 1960.

[18] W. Gatterbauer, P. Bohunsky, M. Herzog, B. Krüpl, and B. Pollak, “Towards domain-independent IE from web tables,” in *WWW*, 2007.

[19] H. Elmeleegy, J. Madhavan, and A. Y. Halevy, “Harvesting relational tables from lists on the web,” *PVLDB*, 2009.

[20] M. J. Cafarella, A. Y. Halevy, Y. Zhang, D. Z. Wang, and E. Wu, “Uncovering the relational web,” in *WebDB*, 2008.

[21] M. Benedikt, G. Gottlob, and P. Senellart, “Determining relevance of accesses at runtime,” in *PODS*, 2011.

[22] M. Benedikt, P. Bourhis, and C. Ley, “Querying schemas with access restrictions,” *PVLDB*, 2012.

[23] J. Madhavan, D. Ko, L. Kot, V. Ganapathy, A. Rasmussen, and A. Y. Halevy, “Google’s deep web crawl,” *PVLDB*, pp. 1241–1252, 2008.

[24] W. Wu, A. Doan, and C. T. Yu, “Webiq: Learning from the web to match deep-web query interfaces,” in *ICDE*, 2006.

[25] X. Jin, N. Zhang, and G. Das, “Attribute domain discovery for hidden web databases,” in *SIGMOD*, 2011.

[26] L. Barbosa, H. Nguyen, T. H. Nguyen, R. Pinnamaneni, and J. Freire, “Creating and exploring web form repositories,” in *SIGMOD*, 2010.

[27] W. Liu, X. Meng, and W. Meng, “Vide: A vision-based approach for deep web data extraction,” *IEEE Trans. Knowl. Data Eng.*, pp. 447–460, 2010.

[28] M. A. Hearst, “Automatic acquisition of hyponyms from large text corpora,” in *ICCL*. Association for Computational Linguistics, 1992.

[29] J. Zhu, Z. Nie, J.-R. Wen, B. Zhang, and W.-Y. Ma, “2d conditional random fields for web information extraction,” in *ICML*. ACM, 2005.

[30] J. D. Lafferty, A. McCallum, and F. C. N. Pereira, “Conditional random fields: Probabilistic models for segmenting and labeling sequence data,” in *ICML*. Morgan Kaufmann Publishers Inc., 2001.

[31] H. Cunningham and D. Scott, “Software architecture for language engineering,” *Nat. Lang. Eng.*, 2004.

[32] N. Kushmerick, “Wrapper induction for information extraction,” Ph.D. dissertation, U. Washington, 1997.

[33] E. Agichtein, L. Gravano, J. Pavel, V. Sokolova, and A. Voskoboynik, “Snowball: a prototype system for extracting relations from large text collections,” *SIGMOD Records*, 2001.

[34] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni, “Open Information Extraction from the Web,” in *IJCAI*, 2007.

[35] F. M. Suchanek, M. Sozio, and G. Weikum, “SOFIE: A Self-Organizing Framework for Information Extraction,” in *WWW*, 2009.

[36] J. Zhu, Z. Nie, J.-R. Wen, B. Zhang, and W.-Y. Ma, “Simultaneous record detection and attribute labeling in web data extraction,” in *KDD*, 2006.

[37] N. Choi, I.-Y. Song, and H. Han, “A survey on ontology mapping,” *SIGMOD Rec.*, 2006.

[38] A. Arasu and R. Kaushik, “A grammar-based entity representation framework for data cleaning,” in *SIGMOD*. ACM, 2009.

[39] X. Liu, X. L. Dong, B. C. Ooi, and D. Srivastava, “Online data fusion,” *PVLDB*, 2011.

[40] http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

[41] A. Deutsch, L. Sui, and V. Vianu, “Specification and verification of data-driven web services,” in *PODS*, 2004.

[42] U. Srivastava, K. Munagala, J. Widom, and R. Motwani, “Query optimization over web services,” in *VLDB*, 2006.

[43] S. Ceri, A. Bozzon, and M. Brambilla, “The anatomy of a multi-domain search infrastructure,” in *ICWE*, 2011.

[44] A. Bozzon, M. Brambilla, and S. Ceri, “Answering search queries with crowdsearcher,” in *WWW*, 2012.

[45] S. Thakkar, J. L. Ambite, and C. A. Knoblock, “Composing, optimizing, and executing plans for bioinformatics web services,” *VLDB J.*, 2005.

[46] K. Q. Pu, V. Hristidis, and N. Koudas, “Syntactic rule based approach to Web service composition,” in *ICDE*, 2006.