Deriving Intensional Descriptions for Web Services

Maria Koutraki, Dan Vodislav, Nicoleta Preda

Deriving Intensional Descriptions for Web Services

Maria Koutraki*, Dan Vodislav**, and Nicoleta Preda*

*DAVID lab, University of Versailles, France
**ETIS lab, University of Cergy-Pontoise, France

Abstract

Many data providers make their data available through Web service APIs. In order to unleash the potential of these sources for intelligent applications, the data has to be combined across different APIs. However, due to the heterogeneity of schemas, the integration of different APIs remains a mainly manual task to date. In this paper, we model an API method as a view with binding patterns over a global RDF schema. We present an algorithm that can automatically infer the view definition of a method in the global schema. We also show how to compute transformation functions that can transform API call results into this schema. The key idea of our approach is to exploit the intersection of API call results with a knowledge base and with other call results. Our experiments on more than 50 real Web services show that we can automatically infer the schema with a precision of 81%-100%.

1 Introduction

Web Services (WS). In recent years, several important content providers such as *Amazon*, *musicbrainz*, *IMDB*, *geonames*, *Google*, and *Twitter* have chosen to export their data through Web services. These services cover a large variety of domains: books, music, movies, geographic databases, transportation networks, social media, even personal data. This trend gained momentum thanks to the Open Data Initiative, the success of mash-up applications, and new initiatives that grant users programmatic access to their personal data. Users can access the data by invoking services, but they can hardly copy the entire content of the remote source. Hence, a Web service seems to be a sweet-spot in the trade-off between sharing the data and protecting it.

The wealth of data exported by Web services is an opportunity for the development of new intelligent applications. For instance, it is possible to develop an application that proposes a trip to a concert. This application could gather concert recommendations from a social Web service such as *Facebook*'s or *Twitter*'s, retrieve offers for concert tickets from a ticketing Web service, check the user's calendar through a Web service from Google Calendar, load train stations from Geonames, and plan the trip through a travel Web service. The central challenge to such an endeavor is that the application has to join data across different services. This is difficult, because each Web service operates under its own local schema.

Modeling Web Services. A Web service is essentially a parameterized query over a remote source. In this paper, we concentrate on REST Web services. These services work by accessing a parameterized URL. For example, Last.fm offers a Web Service which, given an artist as input entity, returns the birthdate, the gender, and the albums of the artist. Figure 2 shows how this service is presented on the Last.fm Web page. We say that *artist* is the *input type* of this Web Service. In order to *call* this Web service, we replace the string "artist" by an *input entity*, says "Frank Sinatra", and access the URL. The server responds by sending the results of the request, which are usually in XML or JSON. Figure 3 shows the result of our example call. It contains the albums of Sinatra with their title, and all their singers, including Sinatra himself. The challenge is now to make this output interoperable with the results from other Web services.

$$\begin{aligned} \mathit{getAlbumsByArtist}^{iooo o}(y_4, y_1, y_2, y_3, y_5) \leftarrow & \mathit{label}(x, y_4), \mathit{birthdate}(x, y_3), \mathit{gender}(x, g), \mathit{label}(g, y_5) \\ & \mathit{created}(x, z), \mathit{label}(z, y_1), \mathit{date}(z, y_2) \end{aligned}$$

Figure 1: View with binding patterns for the Web service getAlbumsByArtist.

```
http://www.last.fm/api/getAlbums?query=artist
Parameters:
   artist (Required): The artist name
```

Figure 2: A REST Web Service

Global Schemas. In order to make the service interoperable with other services, the state of art solution [15] is to assume a common global schema. Then, the Web services are represented as parameterized conjunctive queries (i.e., views with binding patterns) over this global schema. In our example, the global schema could contain relations such as *created* (for an artist who released an album) or *label* (for the relationship between an entity and its name). Expressed in this global schema, the service becomes the query shown in Figure 1. Such a view is essentially a *parameterized conjunctive query*, where y_4 is the *input parameter* (the name of the artist) and all the other arguments are *output parameters* (the birthdate, the gender, the album, and the release date of the album). It is common to distinguish between an entity and its name. For example, while the g will be some internal identifier for a gender, the y_5 will be the label of that gender, such as, e.g., "female". The goal is to have all Web services expressed as views over the same global schema. This way, applications can uniformly reason about Web services in the terms of the global schema.

Problem Description. The central challenge with global schemas is that each Web service has to be mapped into this global schema. While the orchestration of Web services has received much attention lately, the transformation of services into the global schema is often done manually. Our goal is to deduce the schema fully automatically. More precisely, our goal is

Given:
- the URL and the input type of a Web service (Fig. 2)
- a global schema

Compute:
- the view of the Web service in the global schema (Fig. 1)

The URL of the Web Service and the input type can be found by a non-expert user on the Web page of the Web service (as in Figure 2). No knowledge of the output schema of the Web service is required. On the contrary, our goal is to deduce this output schema automatically.

This problem is challenging for several reasons. First, the node labels in the call results are usually vacuous and do not give away any semantics. In the example (Figure 3), they are just "r", "a", and "t". Second, it is not clear which nodes in the call result correspond to entities in the global schema. In the example, one can guess that the nodes labeled with "s" correspond to singers. However, the nodes labeled with "l" correspond to nothing in particular. Third, relations in the global schema can correspond to paths of different length in the call result. For example, the gender of an "s" node is 1 hop away, but the birth date is 2 hops away. Finally, the challenge is to transform a call result (Figure 3) into tuples in the global schema.

Our Approach. Our idea is to build upon existing knowledge bases (KBs). KBs such as YAGO [27] or DBpedia [3] provide a schema, a taxonomy of classes with their instances, and a several million facts in that schema. The instances can be used for probing the Web service. The facts can be used to guess the schema of the Web service. Our contributions are as follows:

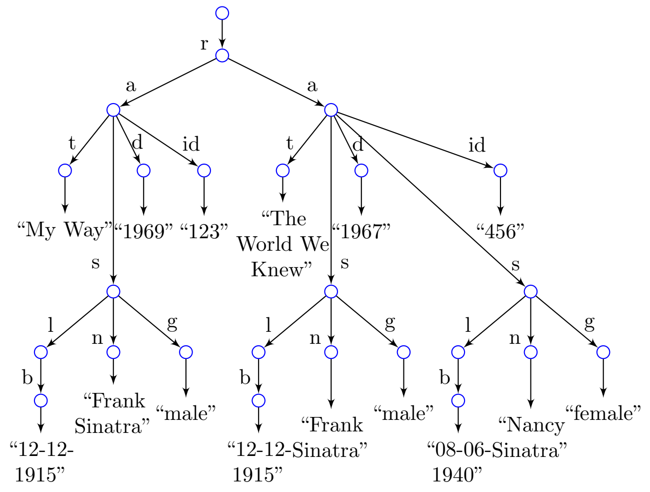

Figure 3: A call result for getAlbumsByArtist

- An algorithm that, given the URL and the input type of a Web service, computes the view in the global schema fully automatically.
- A method to compute a transformation function from the call results into the global schema.
- A variant of our algorithm that can be used to discover hidden input types for Web services.
- Experiments on more than 50 real-world Web services, showing that we can automatically create views with a precision between 81% and 100%, and discover hidden input types with a precision of more than 80%.

The rest of this paper is structured as follows: Section 2 discusses related work and Section 3 describes our data model. Sections 4 and 5 provide more details on the different parts of our method. Section 6 shows a baseline approach to our subject. Section 7 describes our algorithm for mining hidden input types. Section 8 presents experimental results, before Section 9 concludes.

2 Related Work

To the best of our knowledge, no existing work can automatically infer views with binding patterns from Web services. However, our problem shares similarities with schema matching, query discovery, and Web service discovery.

Schema Matching. Determining the description of a Web service in terms of a global schema can be seen as a special case of schema mapping or ontology alignment. This is the problem of matching the concepts and the relations of one schema to the concepts and relations of another schema. Several approaches have been developed to this end.

(*i*) Approaches such as [4, 19] align the schemas of two KBs. In our scenario, we have only one KB, and the other "KB" is the Web service with no formal schema. Even when we call the service, we do not necessarily get information about its schema. This is because the output of the service usually contains only vacuous node labels and no concept names (as in Figure 3). Therefore, schema based approaches cannot be applied in our setting.

(*ii*) Several approaches use the overlap of instances to compute the alignment between two KBs [26, 16]. For us, the instances of the one "KB" is a tree (the call result of the WS) in which some nodes may correspond to entities, and other nodes are just intermediate nodes. The tree does not tell us which nodes are entities and which are not. One solution is to convert every possible path of the tree into a fact. However, as we will see in Section 8, this solution does not work well for Web services. The reason is twofold: First, a node in the tree may actually combine the properties of several entities, which leads the approach ad absurdum. Second, a fact from the call result may correspond to a join of several relations in the KB. State-of-the-art approaches for KB alignment, however, expect a one-to-one mapping.

(*iii*) Other approaches use machine learning to determine a mapping [9, 18]. They require training data, which is either provided by experts [9] or learned from pre-defined mappings [18]. In contrast, our solution requires no such input. We assume no human assistance during the mapping process. The user has no knowledge of the schema of the output.

(*iv*) Works such as [25, 31] aim to bootstrap a global schema. In [25] e.g., the authors show how the schema of a new ontology can be bootstrapped by using as input the WSDL schema descriptions of the Web Services using the SOAP protocol. In our scenario, we target the REST protocol, which provides no such information. Rather, our goal is to map services into an existing schema.

(*v*) We differentiate our work from works that map HTML tables to KBs (e.g. [17, 30]). For tables it is clear that columns are mapped to class entities and pairs of columns to relations. In our case, in contrast, the encoding of relations is unknown.

Query Discovery. Query discovery is concerned with the actual translation of data from one source to another. The state of art solution [19] exploits schema constraints, which are not available for Web services. Closest to our work, [28] derives intensional descriptions for WSs. However, the approach is semi-automatic, and requires the user's assistance during the process. Furthermore, it assumes an implicit translation of call results into tables. It can only deal with WSs that return properties for one class of entities. Our approach, in contrast, can deal with the general case where WSs return nested descriptions of entities of different types.

Web Service Discovery. The approaches of [10, 24] annotate a Web service with the concepts for which it returns instances. Our work goes beyond that. We compute views with binding patterns that map the entire call results to the KB. We also compute transformation functions that enable a uniform access to different Web services and which can be directly used for orchestrating Web services.

Orchestration and Mash-ups. Web service orchestration is concerned with joining several Web services in order to reply to a query [5, 23]. This work does not map services to a common schema. On the contrary, it requires that mapping as input. Other work proposes a new semantic model for representing Web services or mash-ups [22, 29]. Our work, in contrast, aims to represent Web services in the standard model of views with binding patterns [15].

Wrapper induction. Wrapper induction algorithms [8, 12] are concerned with automatically constructing information extraction rules ("wrappers") for Web pages. However, wrapper induction does not map the sources to a global target schema. Our work, in contrast, translates the schema of the WS into the classes and the relations of the KB.

Data Fusion. Given a schema mapping, data fusion [11] is concerned with aligning the instances from several sources. In contrast, our goal is to compute the schema mapping.

3 Preliminaries

3.1 Knowledge Bases

RDF Knowledge Bases. RDF is the standard of knowledge representation on the Semantic Web. The RDF model assumes a set of URIs, a set of literals, a set of classes, and a set of binary relations. A *URI* is

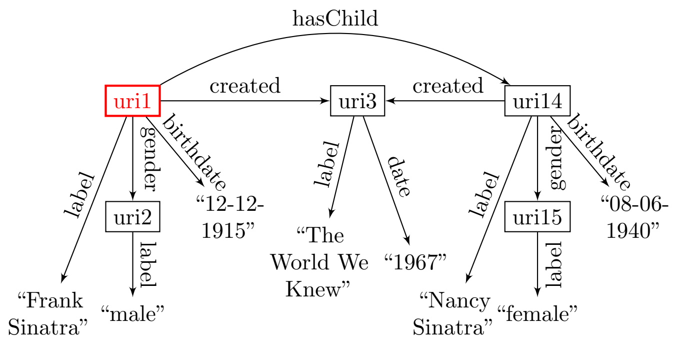

Figure 4: RDF (fragment).

an identifier of a real-world entity such as an organisation, a person or an abstract concept. These entities typically have a *name*, i.e., a human-readable string that identifies the entity. A *literal* is a string, a date, or a number. A *class* corresponds to a set of entities, such as the class of singers or the class of cities. A *relation* holds between two entities or between an entity and a literal. An element of a relation is called a *fact*, and we write $r(x, y)$ to say that the entity with URI x stands in the relation r with the entity with URI y. A *knowledge base* (KB) is a collection of facts. The *domain* of a relation is the class from which all first arguments of its facts are taken. Analogously, the *range* is the class of the second arguments. A relation is called the *inverse* of a relation r, written r^{-1}, if $\forall x, y : r(x, y) \Leftrightarrow r^{-}(y, x)$. We assume that the KB contains all inverse relations and their corresponding facts. This extension results in facts that have literals as their first argument, a minor digression from the standard. It is common to depict a KB as a labeled directed graph, where the nodes are entities or literals and the edges denote facts. Figure 4 shows a snippet of a KB. The entities are shown in boxes while literals are shown in quotation marks.

A relation is called *functional*, if there are no two distinct facts that share the relation and the first argument. Real life KBs may be noisy, and contain distinct facts for relations that should be functional (such as *bornIn*). Therefore, we make use of the functionality [26]:

$$fun(r) := \frac{\#x : \exists y : r(x, y)}{\#x, y : r(x, y)}$$

Here, $\#x : \Phi$ is the number of values for x for which Φ holds. Perfect functional relations will have a functionality of 1.

We assume that the KB schema contains the standard relations *type* (connecting an entity to a class), *subclassOf* (connecting a class to a more general class), and *label* (connecting an entity to a literal representing its name). If the KB contains the facts *type(e, c)* and *subclassOf(c, g)*, then by inference, it also contains the fact *type(e, g)*. We assume that all such inferences have been materialized.

KB Paths. In all of the following, we assume a given KB. We will write $r_1, ..., r_n(x, y)$ to mean that there is a *path* $r_1, ..., r_n$ from x to y that does not visit any node twice:

$$\exists x_1, ..., x_{n-1} : r_1(x, x_1) \wedge ... \wedge r_n(x_{n-1}, y) \wedge |\{x_1, ..., x_{n-1}\}| = n - 1$$

We write $r_1, ..., r_n(x)$ to mean all entities reachable by that path from x, i.e., $r_1, ..., r_n(x) := \{y : r_1, ..., r_n(x, y)\}$. For simplicity, we also use p for a path of one or several relation names.

3.2 Web Services

Web Services. A Web service API provides several *Web services* (WS). A WS can be *called* by accessing a parameterized URL and by retrieving the *call result* document from the server. This document is typically in XML or JSON. For uniformity, we assume that call results in JSON are transformed to XML. This can be done by standard tools. In line with the XML standard, we represent an XML document as a rooted, unordered, labeled tree. We call the elements or the attributes of an XML tree *structure nodes*, and the values of the attributes or the contents of the elements *text nodes*. In Figure 3, structure nodes are depicted as circles while text nodes are shown in quotation marks.

WS Paths. Given a WS f, we refer to the root of the call result for x by $\lambda_f(x)$. We will omit the subscript f if it is clear from the context. We label an edge in the call result with the label of the target node of the edge. In Figure 3, e.g., the topmost edge will be labeled with "r". If an edge leads to a text node, we label the edge with the generic label τ. As for KBs, we write $l_1/.../l_n(x, y)$ to say that there is a *path* $l_1/.../l_n$ with edge labels $l_1, ..., l_n$ between node x and node y. In the example, $r/a/id/\tau(\lambda(FrankSinatra), 456)$. As before, we write p for $l_1/.../l_n$ and we define $p(x) := \{y : p(x, y)\}$.

This gives us a simple language that can express paths both in the call result and in the KB. This language is compatible with SPARQL [1] (the language for querying KBs), with XPath (the language for querying XML documents), and with XSLT [2] (the language used for our transformation functions). To distinguish paths in the call result from paths in the KB, we use "/" for XML and "." for RDF data.

Views With Binding Patterns. We abstract a WS as a *view with binding patterns.* Formally, a view with binding patterns is a conjunctive query of the form

$$q^{\bar{a}}(\bar{x}) \leftarrow r_1(\bar{x}_1), r_2(\bar{x}_2), \ldots, r_n(\bar{x}_n)$$

where $r_1, \ldots, r_n$ are relation names from the global KB and q is a new relation name. The tuples $\bar{x}_1, \bar{x}_2, \ldots \bar{x}_n$ contain either variables or constants. The query must be *safe*, *i.e.* $\bar{x} \subseteq \bar{x}_1, \bar{x}_2, \ldots \bar{x}_n$ (every variable in the head must appear also in the body). Every variable is adorned in $\bar{a}$ with a letter. We use i for input-output variables, and o for output variables. By input-output variable we mean that the result might return new bindings for that variable.

3.3 Observations

A Web service returns information about the input entity. The problem is that this information can be expressed in a multitude of ways in the call results. However, we have noticed that in practice, the call results follow a number of common sense principles.

I. The KB and the WS overlap. If the WS and the KB are from the same thematic domain (or if the KB is a general purpose KB), then we expect to see some overlap between the entities that the KB knows and the entities that the WS knows.

II. The call result is connected to the input entity. A Web service call always contains data related to its input entities. Hence, we expect that the entities contained in the call result appear in the KB and that they are connected to the input entities by paths that do not exceed a certain length. In practice, this length is not larger than three [21].

III. An injective mapping from entities to nodes. Whenever a call result contains an entity, this entity is usually represented as a structured node with sub-nodes. For example, in Figure 3, every album is represented by a subtree that is rooted in an a-node. Hence, we assume that for every entity in call result, there is at least one structured node that is shared by no other entity of the same class. We call this node an *entity node.*

IV. Relations correspond to linear path queries. We observed that typically, a relation between two entities is encoded as a path between the entity nodes and that the sequence of path labels is used consistently, i.e., it always expresses the same relation.

V. Text nodes map to literals in the KB. We have noticed that text nodes typically correspond to literals in the KB. Using a simple string similarity function, we can map a text node to an equivalent literal in the KB in the vast majority of cases.

4 Schema Discovery

Our algorithm for discovering the schema of a WS requires as input (1) a KB, (2) the URL of the WS, and (3) the input type of the WS. We assume that the input type is given as a class of the KB. Our algorithm proceeds in 4 steps:

1. Probing: We call the WS with several entities from the KB and obtain some sample call results (Section 4.1).
2. Path Discovery: We discover paths in the call results from the root to literals (Section 4.2).
3. Path Alignment: We align the paths in the call results with paths in the KB (Section 4.3).
4. View & Transformation Function Construction: Based on the aligned paths, we build the view and the transformation function (Section 5).

We will now detail these steps.

4.1 Probing

Our schema discovery algorithm requires a number of sample call results from the WS. To generate these, we make use of the input type of the WS: We can call the WS with instances of the input type from the KB. For example, if the input type is given as *singer*, we call the WS with instances of the class *singer* from the KB.

Not every instance of the input type in the KB will return a valid call result from the WS. The WS may not know all instances. However, it is likely that the WS stores data for known entities such as famous actors, acclaimed books, or big cities. Our intuition is that the KB, on the other hand, will likely contain more facts about such "important" entities. Therefore, we rank the entities of the input type by the descending number of facts about them, and call the WS with the top k of them. In our experiments (Section 8), we show that a number of $k = 100$ samples is usually sufficient. This corresponds to less than 0.01% of the data that the tested APIs contain.

Some WSs have more than one input. If we encounter a WS with input types $t_1, ..., t_n$, we find "important" entities of t_1. Then we find in the KB the entities of $t_2, ..., t_n$ that are connected to the entity for t_1. Finally, we probe the WS with these n entities. In what follows, we treat the call results as if they were call results about the single input entity from t_1. This will not hurt our mapping algorithm, since the other input entities are connected in the KB. If they are present in the call results, they will be discovered.

The probing gives us a set of input entities for which the calls have been executed and the results have been materialised. We call to such entities *samples*.

4.2 Path Discovery

Given a sample x, our goal is to align paths in the call result that originate at the root $\lambda(x)$ with paths in the KB that originate at the input entity x. For example, for the call result in Figure 3, we want to find that the path $r/a/t/\tau$ connects the root to the title of a song. So we want to align the path $r/a/t/\tau$ in the call result with the path *created.label* in the KB. In the following, we assume a fixed sample input entity x, and we write $(r/a/t/\tau, \mathit{created.label})$ for our path alignment. To find these path alignments, we first generate all paths in the call result from the root to a literal, and all paths in the KB from the input entity x to a literal, and then align these paths.

Generating XML Paths. We first enumerate the paths from the root $\lambda(x)$ to text nodes. The result is a set of pairs of a path p and a text node y, (p, y), $s.t.\ y \in p(\lambda(x))$. This set is easily computed in a depth first traversal of the XML tree. Hence, the computation is linear in the size of the call result.

Generating KB Paths. Next, we generate the paths in the KB from x to literals. We compute the pairs of a path p' and a KB node y' (p', y'), $s.t.$ $y \in p'(x)$ and y' is a literal. As per Observation II, we limit the search to paths of length smaller or equal to three. Hence, the results can be computed in a depth first traversal of the KB graph originating at x.

However, even short paths may lead to a huge number of results. For instance, *livesIn.hasNationality*$^-$ connects a person to all the other persons who have the nationality of the country where the person lives. Hence, the path connects "unrelated" entities via the very general concept *country.* Our goal is to exclude relationships with an absurdly high number of outgoing edges such as *hasNationality*$^-$, *hasYearOfBirth*$^-$, *hasGender*$^-$. For this purpose, we discard paths with relations whose functionality is smaller than $\alpha = 0.1$. In practice, this value proved to be safe as no results were lost.

Path Pairs. The next step generates path pairs that are candidates for the alignment. Two paths are candidates if they select *at least* one value in common. Given the results of the previous steps for the sample, it computes pairs of form

$$(p, p')\ s.t.\ \exists y \in p(\lambda(x)), y' \in p'(x)\ :\ y = y' \tag{1}$$

This means that we implicitly align the input entity x with the root $\lambda(x)$ (Observation II).

String normalization. Note that we need to check the equality of two strings $(y = y')$. However, caution is needed when comparing text values extracted from the XML result to literals in the KB, because the two sources may use different writings. For example, the XML result may contain the string "Sinatra, Frank", while the KB may contain the string "Frank Sinatra". The alignment of strings across such differences is a well-studied field of research. In our implementation, we considered two strings equal if they share the same words, independent of punctuation signs, word ordering, and case. For this purpose, we normalise the values by lowercasing the string, reordering the words in alphabetical order, and omitting punctuation. Let *norm* be the normalisation function. Hence, in Equation (1), the equality $y = y'$ is replaced by $norm(y) = norm(y')$.

Aligning Paths with Common Values. A simple pair-wise comparison of text nodes in the call result to literals in the KB can be highly inefficient. The number of pairs can easily exceed one thousand. Thanks to our normalisation, however, a pairwise comparison for similarity is no longer necessary: two strings are either identical or not considered similar. Therefore, we can use a merge-join algorithm. We sort each of the two sets of tuples produced in the previous steps by the normalised value of the second component. Then we can merge the two lists in order to produce the results. The complexity is $n \times log(n)$ where n is the cumulated size of the two lists. This yields a set of pairs of the form (p, p'), where p is a path from the call result and p' is a path from the KB and both lead to the same value. These are our candidates for the path alignment.

4.3 Path Alignment

For a set of samples S, and a candidate pair of paths (p, p'), we need to compute a confidence score for the alignment of p and p' with respect to S. We present two strategies to this end.

Overlapping

Method. The simplest method to align an XML path p with a KB path p' (which works best in practice) is to use the overlapping of the results. We say that p' *overlaps with* p, if

$$\forall x \in S\ \exists y : p(\lambda(x), y) \wedge p'(x, y)$$

For simplicity, we use y to denote a literal and a text node with the same (normalised) value. An overlap can be stronger or weaker, depending on the proportion of samples that overlap. We compute the *confidence* of the overlap as the number of samples for which their results overlap, divided by the total number of samples:

$$conf(\ (p,p')\ ) := \frac{\#x : \exists y : p(\lambda(x),y) \wedge p'(x,y)}{|S|}$$

Weaknesses of the Method. Although this simple method leads to remarkably good results, some alignments are lost. For instance, consider the alignment: (*r/a/s/l/e/τ*, *diedOnDate*). If the living singers represent more than half of the input entities selected for sampling, then this alignment is lost. Hence, we also experimented with a second strategy for path alignment, which is based on subsumption.

Subsumption

Method. The subsumption strategy aligns two paths if the results of one are included in the results of the other. We are interested in mining KB paths subsumed by XML paths and vice-versa, XML paths subsumed by KB paths. Let p and p' two paths. We say that p' *subsumes* p, if

$$\forall x, y : p(x,y) \Rightarrow p'(x,y)$$

For instance, consider the path pair (*/r/a/s/n/τ*, *label*). The XML path selects the name of the input entity for which albums are returned as well as the names of the other persons who released the respective albums. However, the KB path selects only the name of the input entity. Hence we have: $label \rightarrow /r/a/s/n/\tau$.

Confidence. There are different ways to compute the confidence of such rules [20, 7, 13]. The problem in our case is that our data on both sides is incomplete: When the KB does not contain a certain fact, the WS can still return it, and vice versa. Hence, we opted to use the *PCA confidence*, a measure developed for AMIE [13]. This measure is particularly suited for incomplete relations. The measure assumes that a source knows either *all* or *none* of the p-attributes of some x. Under this assumption, the formula counts as counter-examples for a rule $p(x,y) \rightarrow p'(x,y)$ only those instances x for which the query paths p return an answer, but where y is not among these answers. This yields the following confidence measure:

$$pcaconf(\ (p,p')\ ) := \frac{\#(x,y) : p(x,y) \wedge p'(x,y)}{\#(x,y) : \exists y' : p(x,y) \wedge p'(x,y')}$$

where $\#(x,y) : \mathcal{A}$ is the number of pairs (x,y) that fulfill $\mathcal{A}$.

This formula gives high confidence even if the alignments have small overlap. For instance, in our experiments, we found that the alignment $diedOnDate \rightarrow r/a/s/l/e/\tau$ achieves the highest confidence (*pcaconf*= 1). Hence, we can use *pcaconf* to spot possible alignments even if there is very small overlap. In such cases, the alignment is very risky. Hence, we have to validate such alignments through more probing. This works as follows: Whenever a path alignment achieves high PCA confidence, we select for probing, entities for which the newly discovered KB property is defined. Then we re-probe the WS with these entities.

The result of the path alignment is a set of pairs of the form (p,p'), where p is a path in the call result and p' is a path in the KB. Figure 5 shows the alignment for our running example.

5 View and Transformation

Branching Points. Our next step is to indicate *branching points* in the XML paths and the KB paths. These are denoted by the "*" in Figure 5. A branching point in a path means that there are several outgoing edges in the graph starting from that point of the path that are labeled with the same name. For example, following the path $r/a/t/\tau$ in the call result of Figure 3, we encounter several edges labeled with a. Hence, a

$/r/a^*/t/\tau$	*created*.label*
$/r/a^*/d/\tau$	*created*.date*
$/r/a^*/s^*/l/b/\tau$	*birthdate*
$/r/a^*/s^*/n/\tau$	*label*
$/r/a^*/s^*/g/\tau$	*gender.label*

Figure 5: Results of the path alignment step for *getAlbumsByArtist*

(r/a^*, *created*)	z
($r/a^*/s^*$,)	x
($r/a^*/t/\tau$, *created*label*)	y_1
($r/a^*/d/\tau$, *created*date*)	y_2
($r/a^*/s^*/l/b/\tau$, *birthdate*)	y_3
($r/a^*/s^*/n/\tau$, *label*)	y_4
($r/a^*/s^*/g/\tau$, *gender.label*)	y_5

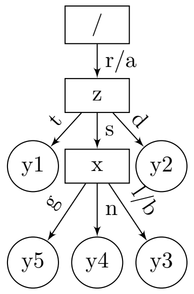

Figure 6: New path alignments and their associated variables for *getAlbumsByArtist* (left) and the tree-like hierarchy of paths in XML (right).

is a branching point. Formally, we say that l_i is a branching point for the XML path $/l_1/l_2\dots/l_{i-1}/l_i\dots/l_n$ *iff*

$$\begin{aligned}\exists x \in S, z, y_1, y_2 \ : \ & /l_1\dots/l_{i-1}(\lambda(x), z)\wedge \\ & \wedge\, l_i(z, y_1)\wedge l_i(z,y_2)\wedge y_1 \neq y_2\end{aligned}$$

We use a path summary in the form of a DataGuide tree [14] to keep track of all possible paths in the samples and of their branching points. Its construction requires only a depth first traversal of each sample tree and its size is small because we do not need to store the text nodes.

For the KB paths, we insert branching points whenever a relation is not a function. Some KBs explicitly declare their functions. If that is not the case, we compute the functionality for each relation (Section 3). Since KBs may contain some noise, we insert branching points at a relation only if its functionality is lower than $\beta = 0.9$.

Cut and Align. We now want to align also sub-portions of the paths. Our input are the path alignments produced by the previous step, annotated with branching points, and our output will be more path alignments. The clue of our algorithm is the insight that if we have a path of functional relations on the KB side, then this path has to be aligned to a (sub-)path with no branching points on the XML side. This means that the rightmost $*$ annotation of one path is aligned with the right most $*$ of the other path in the pair. Let us write $P * p$ to denote a path such that $P*$ ends in a branching point or it is the empty path, and p has no branching points. Then we generate new path alignments by recursively applying the following rule:

$$(P * p, P' * p') \rightsquigarrow (P, P')$$

For the path alignments in Figure 5, this gives rise to the new alignments in the Figure 6 (left).

Weakness of the Method. The annotation with branching points on the XML paths is biased by the set of XML call result samples. For example, if all call results return only singers that have released a single album, then our method will not annotate */s* as a branching point.

Duplicate Elimination. The previous step may result in associating the same KB path to different XML paths. This means that the same relationship appears multiple times in the call result. If this happens, we select the alignment that was generated from the pair with the highest confidence. Also, it may happen that several KB paths map to the same XML path. If the KB paths select literals, we use the same approach. If not, then they concern relationships between complex entities. For instance, we may have as candidates *created* and *bornIn.producedIn*$^{-}$. In that case, we select the path where the maximal functionality of its relations is minimal.

View Definition. We now want to deduce the view definition from our path alignments. We introduce a new variable for each path alignment (shown in the middle in Figure 6). Now, the view definition is simply the join of the KB paths with the appropriate variables. More precisely, let us write $P' \to v$, if some alignment (P, P') is associated to a variable v. Then, we construct the body of the view definition with the following rules:

$$\left.\begin{array}{l} P* \to v_1 \\ P*p \to v_2 \end{array}\right\} \rightsquigarrow p(v_1, v_2)$$

$$\left.\begin{array}{l} P* \to v_1 \\ P*P'* \to v_2 \\ \nexists P''* \to v, P''' : P*P''*P''' = P*P'* \end{array}\right\} \rightsquigarrow P'(v_1, v_2)$$

The resulting atoms are added to the body of the view definition. The head atom of the view definition is the name of the WS. It contains only the variables that correspond to literals. This is because the entity identifiers used by the KB cannot be obtained by calling the WS. For our running example, we obtain exactly the view definition of Figure 1.

Transformation Function. We have shown how to derive automatically the view definition of a WS in the global schema. Now, we turn to the question of how we can transform a concrete call result into tuples of this view definition. Our idea is to use an XSLT script for this purpose. Given a call result of a WS, the script will extract bindings for the variables in the head of the view definition of the WS. We now explain how to derive this script automatically from our path alignments.

We first organise the XML paths of the path alignment as a tree. The root of the tree is labeled with "/", and the other nodes represent variables. Edges are labeled with path queries. In the example of Figure 6 (left), this leads to the tree shown in Figure 6 (right).

Algorithm 1 transforms such a tree into an XSLT script. It requires as input the root node of the tree. We assume that the functions *isRoot(), isLeaf(), getChildren()*, and *pathFromParent()* are available for every node. Let *pathFromParent()* return the path query on the edge between the node and its parent. For leaf nodes, the function omits the last part "text()" (τ). For instance, y_1*.pathFromParent()=t*. Our algorithm performs a depth-first search on the tree, and outputs a variable assignment for each edge of the tree.

The output of this algorithm is an XSLT script. This script is generated only once per WS. Each time a client calls the WS, the client has to execute the XSLT script on the call results, which will yield tuples in the view definition of the WS. We remark that this gives us an end-to-end solution for integrating Web services into a global schema: Given a schema and a WS, we can automatically derive a view definition of the WS, and a method to map the call results into that view definition.

6 Baseline Approach

We have presented an end-to-end solution for the mapping of a Web service into a global schema. We will now present a baseline solution to this problem, which is based on wrapper induction. We adopt the approach that [21] has developed for mapping Deep Web forms into a KB. The approach creates a wrapper that extracts a pseudo-KB on the result pages of the Deep Web form. This pseudo-KB is then aligned with the reference KB using the PARIS ontology alignment algorithm [26]. A similar approach has been presented in [6] for mapping XML documents to RDF ontologies.

Algorithm 1 makeXSLT(TreeNode v)

```
if $v$.isRoot() then
  print <xsl:template match="/">
  for $v'$ : $v$.getChildren() do makeXSLT($v'$) end for
  print </xsl:template>
end if
if $v$.isLeaf() then
  print {$v$:<xsl:for-each select="$v$.pathFromParent()">
  {<xsl:value select="." >} </xsl:for-each>}
else
  print {<xsl:for-each select="$v$.pathFromParent()">
  for $v'$ : $v$.getChildren() do makeXSLT($v'$) end for
  print </xsl:for-each>}
end if
```

In the spirit of wrappers, we construct entities for repetitive structures. In our case, these are subtrees rooted at nodes that can have sibling nodes labeled with the same name. In our running example (Figure 3), these are the nodes labeled with a and s. We use an XSLT script to implement this wrapper. Conveniently, we can use Algorithm 1 to generate this script. As input, we use the DataGuide that we computed for our samples. The branching points and the leaf nodes are annotated with variables. Two XML paths with the same labels are mapped to the same relation in the pseudo-KB. Running this script on the call results will yield a pseudo-KB, much like in [21]. Then, we use PARIS to align this pseudo-KB with the reference KB.

Although this solution may seem reasonable, our experimental evaluation shows that it does not lead to good results. One problem with this approach is that two entities in the reference KB may map to the same entity node in the pseudo-KB. Another problem is that a single relation in the pseudo-KB may correspond to an entire path of relations in the reference KB. This derails KB alignment approaches such as PARIS, which assume that a single relation in one KB corresponds to a single relation in the other KB.

7 Discovering Hidden Input Types

For some WSs the type of the input can be hidden - is defined locally by the Web service. Many API providers use internal ids rather than names to identify entities. Since ids are internal to the API, they can almost never be obtained from external sources (such as the KB). However, if they are contained in the result of some WS of the same API, they can be inferred by joining the data of the two WSs. While some Web services may be joinable, others may be incompatible. For example, it does not make sense to use the output of a WS that delivers albums id as inputs to a WS that expects movies id. We will now show that our approach for mapping Web services into a global schema can also be used to determine which Web services are joinable.

Let ρ_f be a function that maps a variable in the view of a WS f to its bindings in our samples of f. We consider the following problem: Given a WS f_O that has been mapped to the KB, and given another WS f_I with only one input of unknown type, compute join dependencies of the form:

$$(f_O, x, p, f_I)$$

Here, x is variable in the view of f_O. p is an XML path such that $\forall x' \in \rho_{f_O}(x)$, $p(x')$ has values for the input of f_I and f_I returns data relevant to x'.

Example 1 Let f_O=*getAlbumsByArtist* and let f_I be a WS that returns songs by album-id. We aim to discover the following join dependency: ($f_O, z,$*id/*$\tau$$, f_I$). Here, z is the variable used for album entities in the view in Figure 1. In the call result depicted in Figure 3, z had as bindings the two nodes labeled with a. Note that the path *id/*τ selects exactly the ids of the respective albums.

Mini-KB Extraction. The interesting aspect of our example join dependency is that it concerns paths in f_O that were not yet mapped to KB relations (namely *id/τ*). Our goal is now to extend our view definition and our transformation function to these unmapped paths. For each variable x in the view of f_O, and for each XML path p' s.t. $\exists x' \in \rho_{f_O}(x) \wedge p'(x') \neq \emptyset$, we add a new relation name to the schema of the KB, and we map p' to that new relation name. The XSLT script for f_O is updated accordingly. Let *mini-KB(x)* be a KB computed by applying the updated XSLT script to every call result in our sample for f_O.

Generating candidates. A candidate join dependency of the form (f_O, x, p, f_I) is computed for every variable x from the view of f_O and every path p in f_O where

$$\#x' : (x' \in \rho_{f_O}(x) \wedge p(x') \neq \emptyset) > \theta$$

In our experiments, we set $\theta = 20$. For probing f_I, we extract from *mini-KB(x)* tuples of the form:

$$(x^k, l) : \ \exists x' \in \rho_{f_O}(x) \wedge \sigma'_{f_O}(x') = x^k \wedge p(x', l)$$

where l serves to call f_I, but we see x^k as the input entity of that respective call. We now discuss when we keep a candidate join dependency (f_O, x, p, f_I).

Step 1: Catch Error Messages. If f_I returns systematically an error message, we discard the candidate. However, some WSs will not return an error message if fed with a bad input entity. Instead they will return results for all entities whose names or other attributes are similar to the input entity. Therefore, our first step cannot filter out all bad candidate join dependencies.

Step 2: Checking Input's Position in the Output. If the call is valid, then it should return information about the input value. This value should consistently appear under the same path(s) from the root (Observation IV). Assume that we discover that the input value occurs *always* under the path p_i in the call results of f_I. Then, we can filter out every candidate $(f"_O, x", p", f_I)$ that did not discover p_i as being the path with the inputs.

Step 3: Align f_I and the mini-KB. Finally, we use the path alignment algorithm from Section 4 to align label paths in the outputs of f_I with path relations in *mini-KB*(x). We rank the candidates between f_I and f_O according to the number of paths produced by the alignment. We select as solution the top candidate. The intuition is that f_I should return some of the properties of x returned also by f_O. For instance, for our example, we expect that the second WS recalls the singer of the album in its results, which in turn appears also in the output of *getAlbumsByArtist*.

Discovering Types for WS Inputs. Let our candidate (f_O, x, p, f_I) be a join dependency. Hence, f_I returns data relevant to x. Thus, we can say that the input of f_I has as type the class of x^k (or x) in the KB. However, the relation that links x^k to l in the KB might not be the default relation *label*. Hence, the input of f_I is annotated with the class of x^k (its input entity) and the (new) relation to which p is mapped to. With this, f_I is now ready to be mapped to the target KB.

8 Experiments

Our algorithms are fully implemented in a system called *DORIS* (Discovering Ontological Relations In Services).

Data Sources. We have tested DORIS on more than 50 real WSs provided by more than 10 APIs. The data exported by the WSs cover four domains: music, movies, books, and geographic data. Table 1 shows the WS APIs that we considered, grouped by thematic domain. The first column shows the APIs. The third column shows the WSs offered by these APIs. Several APIs can provide the same WS. For example, both

API	No	WS	Path Alignment						Classes		Atoms			
			Overlap		KB→WS		WS→KB		**DORIS**		**DORIS**		*PARIS*	
			P	R	P	R	P	R	P	R	P	R	P	R
GeoData { geonames	1	getCityByName (*City*)	**1**	**0.73**	0.17	0.33	0.0	0.0	**1**	**1**	**1**	**0.67**	0.31	0.67
	1	getCountryByName (*Country*)	**0.62**	**0.71**	0.25	0.2	0.0	0.0	**1**	**1**	**1**	**1**	0.1	0.33
Books { isbndb, library_thing	2	getAuthorInfoById (*Id*)	**1**	**1**	0.29	0.49	0.35	0.89	**1**	**1**	**1**	**1**	0.33	0.37
	2	getAuthorInfoByName (*Author*)	**0.97**	**0.97**	0.32	0.48	0.23	0.5	**1**	**1**	**1**	**1**	0.41	0.62
	2	getBookInfoByTitle (*Book*)	0.84	**1**	0.85	0.72	0.5	0.5	**1**	**1**	**1**	**1**	0.29	0.5
Movies { themoviedb	1	getActorInfoById (*Id*)	**1**	**0.8**	0.33	0.8	0.23	1	**1**	**1**	**1**	**0.67**	0.4	0.67
	1	getActorInfoByName (*Actor*)	**1**	**1**	0.33	0.67	0.25	1	**1**	**1**	**1**	**1**	0.33	1
	1	getMoviesByActorId (*Id*)	0.88	**1**	0.27	0.43	1	0.57	**1**	**1**	**1**	**1**	0.25	0.67
Music { music_brainz, musixmatch, last_fm, echonest, deezer, discogs	6	getArtistInfoById (*Id*)	**0.82**	**0.89**	0.32	0.49	0.33	0.81	**0.89**	**0.78**	**0.83**	0.73	0.38	0.72
	1	getArtistInfoByMBID (*Id*)	**0.57**	**1**	0.2	0.5	0.19	0.75	**1**	**1**	**1**	**1**	0.33	1
	6	getArtistInfoByName (*Singer*)	**0.9**	**0.97**	0.33	0.49	0.46	0.31	**0.89**	**0.94**	**0.84**	0.9	0.28	0.72
	4	getAlbumByTitle (*Album*)	0.84	**0.75**	0.87	0.17	0.2	0.12	**1**	**1**	**1**	**1**	0.09	0.15
	4	getAlbumsByArtistId (*Id*)	**0.94**	**0.99**	0.22	0.2	0.46	0.57	**1**	**1**	**1**	**1**	0.29	0.37
	1	getAlbumsByArtistMBID (*Id*)	**1**	**1**	0.33	0.2	0.5	0.6	**1**	**1**	**1**	**1**	0.5	0.5
	2	getAlbumsByArtist (*Singer*)	**0.85**	**0.91**	0.29	0.31	0.67	0.38	**0.88**	**1**	**0.88**	**1**	0.24	0.33
	1	getTopTracksByArtistId (*Id*)	**1**	**0.93**	0.33	0.13	0.6	0.6	**1**	**1**	**1**	**1**	0.33	0.25
	4	getTrackByTitle (*Song*)	0.81	**1**	0.94	0.53	0.25	0.12	**0.79**	**1**	**0.86**	**1**	0.3	0.32
	2	getTracksByArtistId (*Id*)	**0.88**	**1**	0.17	0.22	0.17	0.33	**0.88**	**1**	**0.88**	**1**	0.25	0.66
	4	getTracksByArtist (*Singer*)	**0.91**	**0.94**	0.29	0.29	0.52	0.41	**0.83**	**1**	**0.81**	0.95	0.23	0.32

Table 1: Path Alignment and Class & Atom Alignment on the YAGO KB.

book APIs support *getAuthorInfoByName*. Therefore, the second column shows how many APIs support a given WS.

We have conducted experiments for three target KBs: YAGO [27] and DBpedia [3] (which are extracted from Wikipedia), and BNF (bnf.fr) (a KB from the domain of books curated by the National Library of France). We probed each Web Service for 100 samples.

Platform. Our system is implemented in Java (JVM 1.7). We use standard Java libraries to parse the XML and JSON call results. We use the Jena 2.11 library to store the KBs, and the Jena SPARQL functionality to query them. We ran all experiments on a personal laptop (2,9 GHz Intel Core i7, 8GB). Our method takes roughly 2 minutes to derive the schema of a WS (Path Alignment and Class & Atom Alignment).

Path Alignment

Our first series of experiments concerns the path alignment (Section 4). We probed each WS with instances of the input domain from the YAGO KB. As a gold standard, we designed the correct path alignments manually for each WS of each API.

Our first goal is to determine the right threshold for the overlapping algorithm. We examine the performance of our method with respect to different threshold values. Table 2 shows the average F-measure across all WSs for the alignment under different thresholds. The threshold 0.5 produces the best results, hence we use it for what follows.

Threshold:	0.3	0.4	**0.5**	0.6	0.7
F-measure:	0.762	0.88	**0.89**	0.88	0.81

Table 2: Average performance of Path Alignment

Then, we ran experiments for both alignment strategies: the overlapping and the subsumption strategy. For the subsumption, we checked whether the results of the KB path are subsumed by the results of the XML path ($KB \rightarrow WS$), and vice-versa ($WS \rightarrow KB$). Table 1 shows the precision and the recall for each WS (averaged over the APIs that provide the WS).

Our experiments show that the overlapping strategy (Overlap columns) can align paths with a precision (P) of 57%-100%. In the vast majority of cases, the precision is over 80%. Recall (R), likewise, is well over 90% in the majority of cases.

The subsumption strategy performs less well. The main reason is that the PCA confidence does not penalise alignments that have the support of a small number of samples. This yields many spurious matches. On the other hand, this allows aligning paths with incomplete relations such as *diedOnDate*, *wasCreatedOnDate*, *hasLongintude*, and *hasLatitude*. In order to find these alignments also with the overlapping method, more samples would be needed.

Class and Atom Alignment

We refer to the bindings of a variable in the view using the term class. By atom we mean an atom of the view. We say that a class or an atom are correctly aligned if the transformation function extracts correct values for them. In order to evaluate the class and atom alignment, we first generated the ideal class and atom alignment manually. Then, we ran our alignment algorithm and compared the results to this gold standard. As input, we used the path alignments computed by the overlapping strategy.

Duplicate Removal. We first wanted to see whether duplicate removal helps precision and recall for the class and atom alignment (see Section 5). Hence, we ran the alignment with and without duplicate removal and measured precision and recall. Our experiments show that the technique helps: By removing the duplicates, the average precision across all Web services for YAGO relations rises from 45% to 91%. At the same time, recall is almost the same. It decreases from 95% for the algorithm that allows duplicates to 93% for the one that removes them. The precision and recall for classes remains unchanged. Therefore, we decided to remove duplicates in what follows.

YAGO. The results of aligning the Web services to the YAGO KB are presented in Table 1 on the right. We are happy to report that our system achieves a perfect precision and a perfect recall on nearly all WSs. Only very few WSs could not be mapped entirely correctly. We reckon that this is the first time that the result of WSs can be mapped fully automatically to the schema of a target KB.

Other KBs. We have also tested our algorithm on DBpedia and BNF. For DBpedia, we used all 50 WSs. For BNF, which is a domain-specific KB, we ran only the WSs from the Books domain. We present an average of the precision and recall of our class and atom alignment in Table 3. Since the BNF contains data that overlaps a lot with the WS, we obtain a perfect precision and recall for this KB. The other alignments are also of very respectable quality, with recall and precision values well over 90%.

	Precision		Recall	
	Classes	Atoms	Classes	Atoms
YAGO	0.92	0.91	0.96	0.93
DBpedia	0.91	0.92	0.98	0.95
BNF	1	1	1	1

Table 3: Average Performance of Alignments

Comparison to Baseline. In Section 6, we introduced a baseline approach to the problem of WS alignment. This approach transforms the WS call results into a pseudo-KB, and uses a state-of-the-art alignment approach (PARIS [26]) to align the pseudo-KB to the reference KB.

We ran PARIS with exactly the same data as DORIS. PARIS computes a confidence score for each alignment. We used a threshold of 0.6 on this score to determine the final output, which was the value for which PARIS performed best. The results of this approach are shown in the last two columns of Table 1. As we see, DORIS outperforms PARIS by a huge margin. DORIS achieves near-perfect alignment, while the precision and recall for PARIS are more in the area of 30%-60%. The reason is that PARIS performs very well when the schemas of the two ontologies are similar. This is not the case at all in our problem. Moreover, PARIS cannot discover complex relations alignments like *hasGender.label*, while DORIS discovered them.

Discovering Hidden Input Types

Our next experiment evaluates our algorithm for discovering join dependencies for WSs whose input type is hidden (Section 7). For each API, we determined the joins manually. Then, we ran our algorithm for each API, and compared the results to the true ones. Table 4 shows the precision and the recall for several variants of the algorithm: S1 and S2 implement Step 1 and Step 2 respectively. S23 implements Steps 2 and 3 together. We find that Step 1 sometimes excludes good answers, which leads to a drop in recall. The good news is that all strategies have a very high recall, which is almost perfect for S2 and S23. Furthermore, S23 achieves a precision that is consistently over 80%. Hence, our method is able to discover join dependencies and input types fully automatically with high accuracy.

API	Precision			Recall		
	S1	S2	S23	S1	S2	S23
deezer	0.21	0.67	0.95	0.77	1	1
discogs	0.27	0.83	0.81	1	1	1
echonest	0.78	1	1	1	1	1
isbndb	0.48	0.82	0.82	1	1	1
last_fm	0.24	1	1	1	1	1
library_thing	0.57	0.6	1	1	1	1
music_brainz	0.1	0.84	0.84	0.93	0.93	0.93
musixmatch	0.11	0.55	0.83	1	1	1
themoviedb	0.19	0.78	0.89	1	1	1

Table 4: Join Dependencies Discovery

9 Conclusion

In this paper, we have shown how to map the results of a Web service fully automatically into the schema of a knowledge base. Our approach aligns the call results of the Web service to relations of the KB, it derives a view definition of the service in the schema of the KB, and it discovers join dependencies between different WSs. Our experiments show that our approach produces near-perfect results on over 50 real Web services in a variety of domains. We hope that our techniques can provide a bridge between different data providers, and thus help the rise of tomorrow's knowledge-centric applications.

Acknowledgments. This work is supported by the research projects EDOP (PATRIMA LabEx) and ALODIS (PEPS FASCIDO 2015 - CNRS).

References

[1] Sparql: `http://www.w3.org/TR/rdf-sparql-query/`.

[2] Xslt: `http://www.w3.org/TR/xslt20/`.

[3] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. DBpedia: A nucleus for a Web of open data. In *ISWC*, 2007.

[4] D. Aumueller, H. H. Do, S. Massmann, and E. Rahm. Schema and ontology matching with coma++. In *SIGMOD Conference*, 2005.

[5] V. Bárány, M. Benedikt, and P. Bourhis. Access patterns and integrity constraints revisited. In *ICDT*, 2013.

[6] I. F. Cruz, H. Xiao, and F. Hsu. Peer-to-peer semantic integration of xml and rdf data sources. In *AP2PC*, 2004.

[7] L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Min. Knowl. Discov.*, 3(1), 1999.

[8] N. Derouiche, B. Cautis, and T. Abdessalem. Automatic extraction of structured web data with domain knowledge. In *ICDE*, 2012.

[9] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD Conference*, 2001.

[10] X. Dong, A. Y. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Simlarity search for web services. In *VLDB*, 2004.

[11] X. L. Dong, E. Gabrilovich, G. Heitz, W. Horn, K. Murphy, S. Sun, and W. Zhang. From data fusion to knowledge fusion. *CoRR*, 2015.

[12] T. Furche, G. Gottlob, G. Grasso, X. Guo, G. Orsi, C. Schallhart, and C. Wang. Diadem: Thousands of websites to a single database. *PVLDB*, 2014.

[13] L. Galárraga, C. Teflioudi, K. Hose, and F. M. Suchanek. Amie: association rule mining under incomplete evidence in ontological knowledge bases. In *WWW*, 2013.

[14] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[15] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 2001.

[16] T. Kirsten, A. Thor, and E. Rahm. Instance-based matching of large life science ontologies. In *DILS Workshop*, 2007.

[17] G. Limaye, S. Sarawagi, and S. Chakrabarti. Annotating and searching web tables using entities, types and relationships. *PVLDB*, 2010.

[18] J. Madhavan, P. A. Bernstein, A. Doan, and A. Y. Halevy. Corpus-based schema matching. In *ICDE*, 2005.

[19] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema mapping as query discovery. In *VLDB*, 2000.

[20] S. Muggleton. Learning from positive data. In *ILP*, 1996.

[21] M. Oita, A. Amarilli, and P. Senellart. Cross-fertilizing deep web analysis and ontology enrichment. In *VLDS Workshop*, 2012.

[22] D. L. Phuoc, A. Polleres, M. Hauswirth, G. Tummarello, and C. Morbidoni. Rapid prototyping of semantic mash-ups through semantic web pipes. In *WWW*, 2009.

[23] N. Preda, F. M. Suchanek, W. Yuan, and G. Weikum. Susie: Search using services and information extraction. In *ICDE*, 2013.

[24] S. Quarteroni, M. Brambilla, and S. Ceri. A bottom-up, knowledge-aware approach to integrating and querying web data services. *TWEB*, 7(4):19, 2013.

[25] A. Segev and Q. Z. Sheng. Bootstrapping ontologies for web services. *IEEE T. Services Computing*, 5(1), 2012.

[26] F. M. Suchanek, S. Abiteboul, and P. Senellart. PARIS: Probabilistic alignment of relations, instances, and schema. *PVLDB*, 5(3), 2011.

[27] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A core of semantic knowledge. In *WWW*, 2007.

[28] M. Taheriyan, C. A. Knoblock, P. A. Szekely, and J. L. Ambite. Rapidly integrating services into the linked data cloud. In *ISWC*, 2012.

[29] P. Traverso and M. Pistore. Automated composition of semantic web services into executable processes. In *ISWC*, 2004.

[30] P. Venetis, A. Y. Halevy, J. Madhavan, M. Pasca, W. Shen, F. W. 0003, G. Miao, and C. Wu. Recovering semantics of tables on the web. *PVLDB*, 2011.

[31] J. Wang, J. Wen, F. H. Lochovsky, and W. Ma. Instance-based schema matching for web databases by domain-specific query probing. In *VLDB*, 2004.