FiLiPo: A Sample Driven Approach for Finding Linkage Points between RDF Data and APIs

Tobias Zeimetz and Ralf Schenkel

Trier University, 54286 Trier, Germany
`{zeimetz, schenkel}@uni-trier.de`

Abstract. Data integration is an important task in order to create comprehensive RDF Knowledge Bases. Many data sources are used to extend a given dataset or to correct errors. Since several data providers make their data publicly available only via Web APIs, such APIs must be included in the data integration process. However, Web APIs often come with limitations in terms of access frequencies and speed due to latencies and other constraints. On the other hand, Web APIs always provide access to the latest data. So far, integrating APIs has been mainly a manual task due to the heterogeneity of API responses. To tackle this problem we present in this paper the FiLiPo (**Fi**nding **Li**nkage **Po**ints) system which automatically finds connections (i.e., linkage points) between data provided by APIs and local knowledge bases. FiLiPo is a sample-driven schema matching system that models API services as parameterized queries. Furthermore, our approach is able to find valid input values for Web API services automatically (e.g. IDs) and can determine not only one-to-one matches but also one-to-many matches. Our results on ten combinations of KBs and APIs show that FiLiPo performs well in terms of precision and recall.

Keywords: Data Integration · Schema Mapping · Relation Alignment

1 Introduction

RDF knowledge bases (KBs) are used in many domains such as bibliographic, medical and biological data. RDF KBs consist of triples, where a triple is a statement of the form (subject, predicate, object). A problem that all databases (regardless of their format) face is that they are potentially incomplete, incorrect or outdated. Considering how much new data is generated daily it is highly desirable to integrate missing data provided by external sources. Thus, data integration approaches [7,6,11,2,10] are used to expand KBs and correct erroneous data. The usual process of data integration is to download data dumps and subsequently align the schemas of the local KB and of the data dump. “Aligning” describes the process by which relations and entities from the local KB are mapped to relations and entities from external sources, thus creating a mapping between local and the external data schemas. Using this alignment, the integration process can be done and the data of the KB is expanded or updated.

However, data dumps are often updated only infrequently. Using live data through APIs instead of dumps [7,6] allows access to more recent data. In addition, the number of potential data sources becomes much larger when using APIs since most data providers share their data not via dumps, but via APIs. According to Koutraki et al. [7], *APIs seem to be a sweet-spot between making data openly accessible and protecting it.* The problems of data integration, i.e. how two different schemas can be mapped, remain. In the worst case, the schema of an external source has a completely different structure than the local KB. Hence, data integration remained a manual task for most parts [7].

Motivation. Connecting KBs with data behind APIs can significantly improve existing intelligent applications. As a motivating example, we consider dblp[1], a bibliographic database of computer science publications. It accommodates different meta data about publications, e.g., titles, publisher names, and author names and is available as an RDF KB. Data from dblp is often used for reviewer, venue or paper recommendation. By extending dblp with information from APIs like CrossRef[2], or SciGraph[3], for example titles or abstracts, these applications can be improved. Missing information about authors like ORCIDs (an ORCID is a code to uniquely identify scientific authors) can be supplemented by these APIs and help to disambiguate author profiles. Furthermore, such information is also useful for a user querying dblp for authors or publications. The aim in this case is to complete the missing information using external data sources. Therefore it is important that multiple APIs can be used and missing data can be integrated from many different sources. Additionally, the determined alignments can be used to identify erroneous data and correct it if necessary.

Contributions. We present FiLiPo (**Fi**nding **Li**nkage **Po**ints)[4], a system to automatically discover alignments between a local KB and APIs, focusing on detecting relational alignments. We omit aligning classes because classes and types do not exist in a typical API response. FiLiPo is designed to work with single response APIs, i.e. APIs that return only a single response and not a list of most similar search results, and works for datasets of arbitrary domains. In contrast to other systems [11], users of FiLiPo only require knowledge about a local KB (e.g. class names) but no prior knowledge about the APIs data. FiLiPo is the first system that automatically detects what information from a KB has to be used as input of an API to retrieve responses. This will not require end users to determine the best input and significantly reduces manual effort. In contrast to other state-of-the-art systems [7], FiLiPo uses fifteen different string similarity metrics to find an alignment between the schema of a KB and that of an API. A single string similarity method is not suited to compare different kinds of data, for example both ORCIDs (requiring exact matches), ISBNs (with some variation) and abbreviated names. Furthermore, FiLiPo is able to find not only 1:1 matches, like other systems, but also 1:n matches, i.e. aligning a single

[1] https://dblp.uni-trier.de/
[2] https://github.com/CrossRef/rest-api-doc
[3] https://dev.springernature.com/restfuloperations
[4] https://github.com/dbis-trier-university/FiLiPo

relation (e.g. full name) with multiple other relations (e.g. first and last name). A user only needs to specify the number of samples sent to the API, rendering the approach usable even for non-technical users such as librarians.

2 Related Work

API Alignment. DORIS [7,8] is the only system that has dealt with the alignment of KBs and APIs so far and builds upon the schema and structure of an existing KB. During the alignment process, the system sends first probing requests to a chosen API. To use DORIS, users only have to specify the input class for the used API and a request limit. The input class specifies which form of entities the API responds to (e.g. Publications). Then the label information of instances of the corresponding class is used as predefined input relation for APIs. However, this is not always the appropriate input for an API; for example, some APIs expect DOIs or ISBNs as input values. Unlike DORIS, FiLiPo is able to detect automatically appropriate input values.

One key assumption of the DORIS system is that it is more likely to find information on well-known, popular or famous entities (e.g. famous actors, acclaimed books, or big cities) via APIs calls than it would be for other entities. Additionally, Koutraki et al. [7] assume that a KB contains more facts (triples) for well-known entities than for lesser-known entities and therefore rank the entities of the input class by descending number of available facts. However, this approach has major drawbacks. For a publication, for example, the number of facts stored by a bibliographic KB is often determined by the meta data of that publication, not by its popularity (unless citations etc. are stored). Furthermore, DORIS uses equality (ignoring punctuation and case) to compare data values in the alignment process. As stated previously, a single method is not suited to compare all various data types. The limitations of DORIS become clear when examining, for example, author names or titles. Names are often abbreviated and the matching approach of DORIS will fail because DORIS performs an exact match on the normalised names (i.e., removed punctuation). Similar problems will arise when examining titles. In contrast to this approach FiLiPo uses a set of fifteen similarity methods and picks randomly chosen entities of a KB.

The problem of aligning the schema of KBs and APIs shares similarities with various fields [12,2,7] like schema matching, data warehouses, e-commerce, query discovery and Web service discovery. Hence, insights and procedures of systems from other fields were also taken into consideration when developing the FiLiPo system. Aligning data of local KBs with that of APIs has similar problems as schema mapping or ontology alignment. The major difference to schema and ontology alignment is that API responses do not always have clear semantics or any semantics at all. In addition, API responses usually do not provide information about classes and relations that can be used during the alignment process. When using APIs, only instance information is available and therefore classical schema and ontology approaches are not suitable for this task. In addition, Madhavan et al. [9] state that KBs often contain multiple schemas and data models to materialise similar concepts and hence build variations in entities and their relations. This makes purely schema-based matching inaccurate, which must therefore be supported by evidence in form of instances from the KB.

Instance-Based Alignment. Instance-based alignment systems use the information bound to instances in KBs in order to find shared relations and instances between two KBs. These approaches can be divided into instance-based class alignment approaches and instance-based relation alignment approaches. The main difference between class and relation alignment lies in the fact that relations have a domain and range. Even if relations share the same value, they can have different semantics (e.g. `editor` and `author`).

A lot of works [11,3,9,5] focus on instance-based relation alignment between two KBs. However, most of them focus on finding 1:1 matches, e.g. matching `publicationYear` to `year`. The iMAP system [3] semi-automatically determines one-to-one matches, but also considers the complex case of 1:n matches. The iMAP systems consists of a set of search modules, called searchers. Each of the searchers handles specific types of attribute combinations (e.g. a text searcher). FiLiPo follows a similar approach. Instead of searchers, FiLiPo only distinguishes between the type of information (numeric, string, or is it a key). Then, in case of strings, a number of different similarity methods are used, and the best method is automatically determined and used.

Similar to iMAP, MWeaver [11] also needs user assistance. MWeaver realises a sample-driven schema mapping approach which automatically constructs schema mappings from sample target instances given by the user. The idea of this system is to allow the user to implicitly specify mappings by providing sample data. However, this approach needs significant manual effort. The user must be familiar with the target schema in order to provide samples. In contrast to this approach, FiLiPo draws the sample data randomly from the knowledge base and thus tries to cover a wide range of information from the knowledge base.

SOFYA [5] is an instance-based on-the-fly approach for relation alignment between two KBs. The approach works with data samples from both KBs in order to identify matching relations. The core aspect of SOFYA is that the standard relation “sameAs” is used to find identical entities in two different KBs. However, this mechanism cannot be used for the alignment of KBs and APIs, because RDF KBs do not contain sameAs links to APIs.

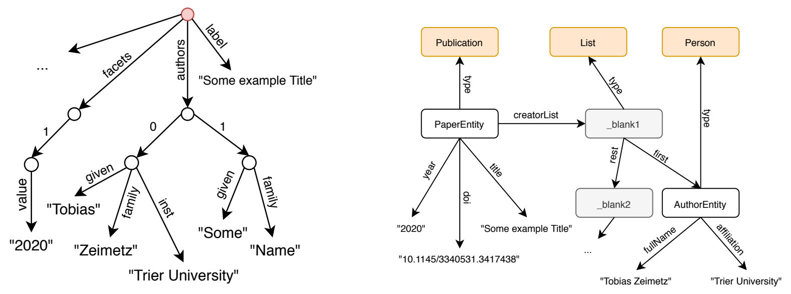
The Cupid system [10] is used to discover an alignment between KBs based on the names of the schema elements, data types, constraints and structure. It combines a broad set of techniques of various categories (e.g. instance-based, schema alignment, etc.). The system uses a linguistic and structural approach in order to find a valid alignment. Furthermore, Cupid leverages a corpus of schemas and mappings to improve the robustness of the schema matching algorithms. Unfortunately, this approach cannot be used when aligning KBs and APIs, since often there is no formally defined schema for an API.

3 Preliminaries

This section introduces RDF knowledge bases and corresponding terms and then provides foundations of APIs.

3.1 Knowledge Base

RDF Knowledge Bases. The RDF format has established itself as the standard for representing and storing KBs. An RDF KB can be represented as a graph with labelled nodes and edges. The edges in RDF KBs correspond to triples of the form $t = (s, p, o)$. Such an RDF triple represents a fact in the KB. The subject s describes the entity the fact is about, e.g. a paper entity. The predicate p describes the relation between the subject and the object, e.g. `title`. The object o describes an entity, e.g. an author of the publication, or is a literal, e.g. the title of a publication.



(a) API Response Example. (b) Fragment of an RDF KB.

Fig. 1: Record of a KB and the corresponding API response.

A class in a KB is an entity that represents a group of entities. Every entity contained in a KB is assigned to at least one class. Entities assigned to a class are denoted as an instance of the assigned class. In Figure 1b, the entity `PaperEntity` is an instance of the class `Publication`.

Relations. Since this paper focuses on aligning relations, we introduce a formal definition for relations and relation paths in a KB. In the following we assume that we have given a KB K. If $(s, r, o) \in K$, we say that s and o are in relation r, or formally $r(s, o)$; in other words, there is a path from s to o with label r. Additionally, we write $r_1.r_2.....r_n(s, o)$ to denote that there exists a path of relations $r_1, r_2, ..., r_n$ in K from subject s to object o visiting every intermediate node only once. For example, in Figure 1b the relation `year(PaperEntity,"2020")` describes the path from the entity `PaperEntity` to the value `"2020"`. In the following we will refer to $r_1.r_2.....r_n(s, o)$ as relation-value triple.

Identifier Relations. Some KBs contain globally standardised identifiers such as DOIs, ISBNs or ISSNs. Such identifiers are only bound to a single entity

and should therefore be unique. Therefore, relations r that model identifier relations have the constraint that their inverse relations (r^{-1}) are "quasi-functions", i.e., their inverse relations have a high functionality. Many works [7,13,4] have used the following definition for determining the functionality of relations:
$fun(r) \coloneqq |\{x : \exists y : r(x, y)\}| \div |\{(x, y) : r(x, y)\}|$

Since real world KBs are designed and modeled by humans identifier relations are often error-prone and noisy. For this reason some identifier values may appear more than once. Therefore, we consider every relation r contained in K with $fun(r^{-1}) \geq \theta_{id}$, where $\theta_{id} \in [0, 1]$ is a threshold, as *identifier relation*.

Knowledge Base Schema. In order to capture the rough structure of a KB a schema enriched with statistical information and more is needed. Ontologies only contain information about classes, sub-classes and relations. However, a KB contains more information hidden in its instances than in a simple ontology, e.g. the number of instances of a specific class, the number of occurrences of a relation with a specific class, or what classes are connected to each other. For the rest of the paper we will refer to a schema $S = (C, R_s, R_{id})$ of a KB as a triple of three sets. The first component C defines the set of classes and how many instances of each class exist in the KB. For the fragment of a KB presented in Figure 1b an example of an entry in C would be a tuple of the form (`Publication`,1). The second component R_s represents the structure of the KB. It contains information about how classes and literals are connected to each other, how often a class is connected to another class and how often a relation is used by a specific class (e.g., $(\texttt{Publication}, \texttt{creatorList}, \texttt{List}, 1) \in R_s$). The last component is the set R_{id} which contains all identifier relations contained in the KB (e.g., `doi`).

3.2 Web Services

Web API. A Web service can provide one or multiple APIs to access data. APIs are called via parameterised URLs responding with a document. As shown in Figure 1a the response of an API is typically an unordered and labelled tree. Inner nodes in the tree represent an object (similar to an entity in a KB) or an array, leaf nodes represent values. The path to a node represents a relation between an instance (similar to an entity in a KB) and another instance or value. To avoid confusion we will describe the relations in a response only as paths.

Path-Value-Pairs. In order to find valid alignments between KBs and APIs the information in the API responses has to be compared with the values of the corresponding entities in a KB. Since comparing objects and arrays from the API response with entities from the KB to determine alignments is not promising, only paths to leafs (literals) have to be considered. Given an API response res we will write $p_1.p_2.....p_n(o)$ to denote that it exists a path $p_1, p_2, ..., p_n$ in res from the root of the response to the leaf o with these labels. For example, in Figure 1a the path `label("Some example Title")` describes the path from the root of the API response to the leaf `"Some example Title"` via the path `label`.

Branching Points. A branching point in an API response indicates that there are several outgoing edges from one node, labelled by numeric index values 0 to n. These branching points represent arrays in the response. For example the path `authors.0.given("Tobias")` in Figure 1a contains a branching point. This indicates that all array entries carry the same information and should therefore be mapped to the same relation. To indicate a branching point, we will use the symbol `*` instead of the numeric index in paths; in the example, we will write `authors.*.given("Tobias")`. Using the same logic, a relation in a KB that points to a set of entities is considered to be a branching point (e.g. `creatorList` in Figure 1b). Additionally, we write $P*$ to indicate a path $P * p$ that has P as prefix and p as suffix, with a branching point separating the two parts.

4 Problem Statement

FiLiPo operates in two phases. In the first *probing phase* FiLiPo sends various information (e.g. DOIs, titles, etc.) to an API to determine which information the API responds to. Then, in the second *aligning phase*, the information returned is used to guess the schema of the API and to determine an alignment between the local and external data. We will now discuss some of the major problems that FiLiPo encounters and solves in more detail.

Probing Phase. The first major challenge occurs in the probing phase. The goal of the probing phase is to determine which input values (e.g. DOI, ISBN, etc.) have to be sent to the API to retrieve a valid response. When a resource is requested that is unknown to the API, it can respond in several ways. The classic and simple case is that it returns a corresponding HTTP status code (e.g. `404 Not Found`). The more complicated case is when the API simply replies with a JSON response that contains an error message, or even returns information on a "similar" resource (e.g., with a similar DOI). This cannot be easily distinguished from a "real" response which contains data about the requested resource.

Alignment Phase. In the alignment phase, valid matches between the data of an API and a KB are determined by comparing the data of the API response with the data of the KB. The first challenge in this context is that the same value may be represented slightly differently in the KB than in the API response (e.g., names with and without abbreviated first names), so this comparison needs to apply string similarity methods. The various existing similarity methods have different strengths and weaknesses. For example, Levenshtein distance is good for comparing the titles of a paper or movie, but performs poorly when comparing names of authors or actors because names are often abbreviated and first and last names may be in different order. Therefore, the best performing similarity method needs to be determined automatically for each type of data.

A special case of this challenge is comparing identifiers such as DOIs and ISBNs. For example, DOIs of a conference book and conference papers are very similar, in many cases only the last two digits will be different. In contrast, the ISBN of a book can be written in different forms (e.g. 978-3-89318-084-4 or 9783893180844) but should be considered equal. For this reason a simple check for equality is not sufficient, otherwise possible alignments are lost.

Finding a match between the records of a KB and an API response can be particularly problematic if the API responds with a record similar to the requested one if that is not found. For example, a request for a book with title “Some example Title” may lead to an API response with information of a book with title “Some Title”. The data of both records may overlap, especially for data with a low functionality such as years that appear in many entities. Thus, a system has to check if the API has responded with the requested record. Koutraki et. al [7] state that if the KB and the API share the same domain, it is likely that the data of the requested KB entity and the API record overlaps. This means that if the information in the records overlaps sufficiently, the API has probably responded with the requested record and not with another record.

One observation we made during the development of FiLiPo is that some values, such as years, are included in the records several times. This is because they represent different information. For example, some bibliographic APIs respond with data containing references and citations of a paper, which often include the publication year. When the publication year of the KB is matched with the publication year of a reference (e.g. because the reference was published in the same year), the matching is no longer correct. Theoretically the semantics of the paths should be considered but as mentioned earlier, API responses do not always have a clear or a directly resulting semantics. The problem becomes even worse when considering the modelling process of the API or KB schema. The value stored for the publication year of a paper often depends on the opinion of the data provider. Some data providers assume that the publication year is the year the conference proceedings are published. Other data providers model the year of the corresponding conference as publication year of the paper, since the paper was presented in this year at the conference.

5 Schema Matching and Mapping

In the following we describe the probing and aligning phase in more detail. The input to the aligning process is the URL of the API and the corresponding input classes in the local KB, where an input class is a class of entities that will be used to request the API.

5.1 Probing Phase

The probing phase is used to find the set R_{in} of relations of the input class that point to values which can be used to request the API successfully (e.g., a DOI relation), since not all input values will lead to responses. To illustrate this with an example, we assume that the input class of the API whose result is presented in Figure 1a is `Publication`. The illustrated fragment of a KB in Figure 1b has five relations to describe the metadata of a publication but the API only responds to DOIs. First, all relations that are not connected to literals (e.g., `type`) are ignored. Afterwards some initial requests are sent to the API for each remaining relation of the input class in order to understand which relation provides successful input values for the API.

The input values for each relation (e.g. DOIs) that are used to request the API are picked uniformly at random from entities of the input class in the KB with this relation. This is done to prevent the entities from being very similar to each other and thus increase the probability of an API response. For example, assume that the API in Figure 1a only responds to records with a specific publisher like Springer. If the entities are selected in any non-random way, e.g. according to the amount of facts the KB contains about them, it is possible that no entities with publisher Springer are included, and the API cannot answer the requests and no aligning can be done.

After sending a request to the API it can respond in several ways. In the best case, the API responds with the HTTP status code `200 OK` or with an HTTP error code (e.g. `404 Not Found`). In the worst case, the server responds with a document containing an error message. In this case the system cannot easily detect that some input values did not lead to a (successful) response and therefore will continue with the alignment phase with the corresponding relation. Since this would result in a considerable increase of requests and runtime it is important to identify error messages.

In order to identify error responses, the system iterates over all answers and compares how similar they are to one another. This procedure is based on the observation that error responses are always similar or even the same, i.e. they usually contain the same error message or consist of an generic error message in combination with the request value. In contrast, correct answers are different to one another since they contain information about various different entities. As a result, an error response is determined by counting how often a response was similar (by using Levenshtein) to other responses. The one that is most often similar (i.e. the similarity is higher than 0.80) to other responses is considered an error message. Then all responses similar to this response will be deleted and all relations r_{in} which have not received enough answers will no longer be considered as valid input relations. In this way unnecessary requests are prevented.

Next, the alignment phase begins, considering only the set R_{in} of relations that led to valid answers. The aligning phase itself is divided into two parts: (1) determining candidate alignments and (2) determining the final alignments.

5.2 Aligning Phase: Candidate Alignment

The candidate alignment phase takes as input the set of valid input relations R_{in}, the chosen KB K and the corresponding schema $S = (C, R_c, R_{id})$. In the following we write $S.C$ to indicate that we access the C component of S (analogously for the other components). For each input relation $r_{in} \in R_{in}$, the algorithm sends n_r further requests to the API.

These requests are constructed similarly to the probing phase. A random entity e is chosen from the input class. FiLiPo then retrieves the set rec of all facts that K contains for e in the form of relation-value triples $r(e, l)$. Like Koutraki et al. [7] we take all facts into account up to depth three. This depth was chosen because all other facts usually do not make statements about the entity e. To exclude the case that entities are connected to other entities in only one direction, inverse relations are also considered. Afterwards FiLiPo calls the API with values v_{req} of the input relation r_{in} of e and stores the response in res.

The next step is to find all relation matches R between the fact set rec and the response res. The set rec contains relation-value triples of the form $r(e, l)$ and res encodes information from the response as path-value pairs of the form $p(v)$ where p is the path in the response from the root to the value v. To realise this, all values l of the local record rec must be compared with all values v of the API response res. For each such pair $(r(e, l), (p(v))$, the best similarity method is determined. If l or v is an IRI, it is important that they are compared with `equals` as IRIs are identifiers and therefore only the same if they are identical. The same holds for numerical values since it is unclear how to check numbers for similarity. In all other cases FiLiPo uses a set M_{sim} of fifteen different similarity methods[5] with several variants since one string similarity method is not sufficient to compare all different data types. The method $m \in M_{sim}$ that returns the largest similarity of l and v is considered (temporarily) to be the best method to compare both values and is stored for the later process.

As discussed before, fuzzy similarity measures are not appropriate for identifier relations, but comparing them for equality would be too strict; identifier relations contained in $S.R_{id}$ are therefore compared with a gradient boosting classifier working on Flair [1] embeddings. We use Flair embeddings instead of others since this framework is character-based and therefore suits better for the comparison of two identifier values. Once the best similarity function has been determined, and if this function yields a similarity of at least θ_{str}, the triple (r, p, m) is created and added to the set of record matches R.

If enough relation matches are found, it is assumed that the input entity e and the API response overlap in their information and that the API has actually responded with information about the requested entity. If the overlap is greater than θ_{rec}, the overlap is considered sufficient and the matches R will be added to $A_{r_{in}}$. This set represents the set of matches found for the input relation r_{in}. If not enough matches are found, it is assumed that the API has responded with information of a different entity; in this case, any matches found between the records must be ignored.

5.3 Aligning Phase: Final Alignment

Afterwards the set $A_{r_{in}}$ is used to determine the final alignment from the temporary matches. For each relation in $A_{r_{in}}$ the valid path match on the API side is searched (if existing). It is easy to match relations and paths without branching points, e.g., `label("Some example Title")`. Here we can simply match `label` with `title`. However, for matches with a branching point path, we need to decide if all entries of the corresponding array provide the same type of information or different types. In the first case, e.g., an array specifying the authors of a paper, we need to match *all paths* that are equal (with exception of index values)

[5] All used similarity methods are listed in our manual at https://github.com/dbis-trier-university/FiLiPo/blob/master/README.md

of the API response with the same relation. This is the case in the example in Figure 1a for the path `authors.*.inst`. In the last case, where every entry of the array has a different type, each different index value at the branching point should be mapped to one specific relation, possibly different relations for the different index values. In the example, `facets.0.value("2020")`) always denotes the year of the publication, whereas `facets.1.value("Computer Science")`) (not shown in the example) could denote the genre of the publication. Therefore, matching either the year or the genre relation of K to `facets.*.value` is incorrect and should be prevented. As an orthogonal problem, some relations in the KB (like `fullName` in the example in Figure 1b) have to be matched with multiple paths from the API response (1:n matches), e.g. `given` and `family` in Figure 1a, because the schematic structure differs.

In order to determine the final alignment and to solve the problems mentioned above, FiLiPo distinguishes three cases: (1) 1:n matches, (2) fixed path matches and (3) branching point matches. First, for every relation r for which at least one tuple (r, p, m) exists in $A_{r_{in}}$ we determine the path $P*$ (index values are replaced by the wild card symbol) that was matched most often in $A_{r_{in}}$, regardless of which method m was used. Furthermore it is important that the relation r and the path $P*$ are matched sufficiently frequent. If such a path was found only a few times, e.g. once in 100 responses of the API, then the match is not very strong. We therefore calculate a confidence score for the matching by dividing the number of valid matches for r by the number of responses. This confidence must be greater than θ_{rec}. We reuse θ_{rec} here based on the assumption that the overlapping of records is also reflected in the overlapping of relations.

To identify 1:n matches, e.g. matching the paths `authors.*.family` and `authors.*.given` to `creatorList.*.fullName`, we assume that the family and given path could be matched similarly often to `creatorList.*.fullName`. In order to identify such matches, we search all entries $B \subseteq A_{r_{in}}$ with the same relation r and the same prefix in the corresponding branching point path $P'*$ which could be matched similarly often to r as the previously determined path $P*$ (with a tolerance). We assume that information representing a 1:n match (e.g. given and family) belong together in principle and are therefore located at a similar position in the result tree of the API. Therefore, an additional condition is that 1:n matches must have the same prefix (up to the branching point), e.g. `authors.*`. If this is the case, all identified paths for r are considered as 1:n matches. Before adding these matches to the set of final alignments, as last step the index values for all found paths are examined. Since a path of an API response can contain more than one array index as nodes, it is necessary to check where the path really branches off. An example is a path of the form $record.0.author.*.name$. The API in this example always responds with an array `record` containing one element, which in turn contains the response. Therefore, it would be inaccurate to model the first array as branching point. To find out at which point a path really branches, the longest common prefix is determined.

If $B = \emptyset$, then only one match was found for the relation r, so it is no longer considered a 1:n match. The next step is to check if the pair $(r, P*)$ is a branching point match or a fixed path match. Therefore, it is checked if the corresponding path $P*$ that was matched to r in $A_{r_{in}}$ only had one index value at the branching point or if multiple different ones where used. An example is the path `facets.0.value` in Figure 1a. To match the relation `year` there is always only this fixed path, because the first place of the array `facets` always describes the year of publication and never contains any other information. If only one index value is found, it is not considered a branching point but a fixed path match. To ensure that it is a valid fixed path match, the confidence for this match is determined as before. If the confidence is greater or equals than θ_{rec}), it will yield a valid fixed path match and the relation-path match is added to the final alignment set.

Some relations and paths are dependent on the previous entity. For example, to match the affiliation path for an author we have to include the whole author array of the API response because matching only one specific path (e.g. `authors.0.inst`) would not be sufficient. Therefore, if more than one index value was found for the branching point $P*$ it is possible that $(r, P*)$ is a branching point match. A match of a relation r and a branching point path $P*$ is considered valid if the following two conditions are satisfied: (1) if the relation r has led to a match often enough, i.e. the previously computed confidence value is greater than or equal to θ_{rec}, and (2) if the matched path $P*$ occurs frequently enough in all matches with the relation r. If both conditions are met, the match between r and $P*$ is considered a branching point match and added to the final alignment set. Note that before adding, the longest common prefix will be determined as in the case of 1:n matches.

For the sake of simplicity, one aspect has not yet been considered in detail. Some relations can also potentially be matched with multiple paths in the API response. For example, the relation for the publication year could be incorrectly matched with the path to the publication years of the article's references. To mitigate such errors, a reciprocal discount is used, i.e. the number n of matches found for a possibly incorrect path p and a relation r is discounted by the length difference of the paths to $n/|(len(r) - len(p))|$. Thus paths with the same length as the KB are preferred. At the end the final alignment set contains all valid matches found for the input relation r_{in}.

6 Evaluation

Baseline System. Only DORIS [7] has dealt with the alignment of KBs and APIs so far. Additionally, many of the systems [11,3] presented in Section 2 work semi-automatically with user assistance and are mostly designed for data sets of the same format. Some of the systems [10] exploit schema information, use semantics or "sameAs" relations to find alignments. However, schema information exists very rarely on the API side and using semantics or relations is difficult because API responses do not always have clear semantics. Furthermore, "sameAs" predicates are a concept of RDF and are not present in classical API responses. Thus, we use only the DORIS system as a baseline system.

Datasets and Platform. We evaluated FiLiPo[6] and DORIS on three local KBs, seven bibliographic APIs and two movie APIs. The first local KB is an RDF version of the dblp[7]. The other local KBs are the Linked Movie DB[8] and an RDF version of IMDB[9], both containing movie information. The used APIs are SciGraph [10], CrossRef [11], Elsevier[12], ArXiv[13], two APIs provided by Semantic Scholar[14] (one with DOIs and one with ArXiv keys as input parameters) and the COCI API of Open Citations[15]. All of these APIs respond with metadata about scientific articles. To align the movie KBs we used the APIs of the Open Movie Database (OMDB)[16] and The Movie Database[17]. It responds with metadata about movies, e.g. movie director and movie genres. All experiments are done on a workstation (AMD Ryzen 7 2700X, 48GB RAM) and all KBs are processed and stored as triple databases by using the Apache Jena Framework.

As a gold standard, we manually determined the correct path alignments for each API. Table 1 shows (column HA) how many valid alignments exist. Alignments were ignored that could not be determined based on the data, but for which a human would have been able to draw a connection. For example sameAs relations cannot be determined automatically since the URLs may differ completely. However, a human could find such a match.

Evaluating Thresholds. FiLiPo works with two different thresholds: the string similarity θ_{str} and the record overlap θ_{rec}. To determine a combination of both thresholds that provides good alignment results, we tested all combinations of values for both thresholds (steps of 0.1) with the CrossRef API and calculated precision, recall and the F1 score. We observed that the found alignments had a very high precision for θ_{str} between 1.0 and 0.5; recall was significantly better at 0.5. This is mainly due to the fact that data which are slightly different (e.g. names) can still be matched. For large values of θ_{rec}, many alignments are lost, because the data of a local KB and an API overlap only slightly in the worst case. Here, a value of 0.1 to 0.2 was already sufficient to prevent erroneous matching. For this reason, we used $\theta_{str} = 0.5$ and $\theta_{rec}0.1$ in the experiments.

FiLiPo Evaluation. We assume that users of the FiLiPo system are non-technical users without programming knowledge or technical skills. Furthermore, users have no in-depth knowledge of external data sources, but are familiar with the structure of the local KB. We assume that users have domain knowledge and therefore can understand common data structures from the genre of the local

[6] https://github.com/dbis-trier-university/FiLiPo
[7] provided by dblp: https://basilika.uni-trier.de/nextcloud/s/A92AbECHzmHiJRF
[8] http://www.cs.toronto.edu/~oktie/linkedmdb/linkedmdb-18-05-2009-dump.nt
[9] https://www.imdb.com/
[10] https://scigraph.springernature.com/explorer/api/
[11] https://www.crossref.org/services/metadata-delivery/rest-api/
[12] https://api.elsevier.com
[13] https://arxiv.org/help/api
[14] https://api.semanticscholar.org
[15] https://opencitations.net/index/coci/api/v1
[16] http://www.omdbapi.com
[17] https://developers.themoviedb.org/3/find/find-by-id

Table 1: Total Requests (TR), Average Probing Time (APT), Average Alignment Time (AAT), Average Number of Alignments (AA), Mean Precision (MP), Mean Recall (MR), Mean F1 Score (MF1), Alignments (A), Precision (P), Recall (R)

	FiLiPo							**DORIS**			
Data Sets	TR	APT	AAT	AA	MP	MR	MF1	A	P	R	F1
dblp ↔ CrossRef	750	18.0	4.0	18	1.00	0.85	**0.92**	9	0.89	0.36	0.51
dblp ↔ SciGraph	750	14.5	2.5	18	0.96	0.78	**0.86**	11	1.00	0.38	0.55
dblp ↔ S2 (DOI)	750	24.5	8.0	15	0.89	0.87	**0.88**	12	0.83	0.47	0.60
dblp ↔ S2 (ArXiv)	750	24.5	9.0	7	1.00	0.88	**0.94**	6	0.83	0.33	0.47
dblp ↔ Open Citations	750	23.0	19.0	16	1.00	0.78	**0.88**	9	1.00	0.33	0.50
dblp ↔ Elsevier	1050	17.5	5.5	13	0.92	0.92	**0.92**	13	0.85	0.85	0.85
LMDB ↔ TMDB	225	4.5	2.0	6	0.94	1.00	**0.97**	7	0.57	0.80	0.67
dblp ↔ ArXiv	100	-	3.5	8	0.83	0.86	**0.85**	5	1.00	0.43	0.60
LMDB ↔ OMDB	100	-	3.5	14	0.93	0.95	**0.94**	11	0.55	0.56	0.55
IMDB ↔ OMDB	100	-	40.0	9	0.73	0.66	0.69	9	1.00	0.90	**0.95**

database (e.g. bibliographic meta data). In addition, experts with knowledge of the APIs (technical-user) can make further settings (e.g., changing string similarity thresholds) to fine-tune the system. Therefore, we divided the APIs into two sets: (1) non-technical evaluation set and (2) technical evaluation set. The non-technical set contains CrossRef, SciGraph and Semantic Scholar (with the two different APIs). All APIs from the non-technical evaluation set were executed with the default settings of FiLiPo, i.e. 25 probing requests with 75 additional requests for valid input relations, thus 100 requests are made for every valid input relation. Since dblp contains only relatively few publications published by Elsevier, we set the number of additional requests for Elsevier to 375. With only 100 requests, the API returned no or few responses.

The technical-user set consists of ArXiv and OMDB. ArXiv was chosen for this set because it always responds with a list of the top most similar entities to the requested one. The only exception is when the API receives an ArXiv key it will respond with a single record. Since FiLiPo is at this stage not able to pick the requested record from a list of top-k similar records the probing phase will not be successful. Because we assumed that technical-users have knowledge of the used APIs, they are able to provide a valid input relation (i.e. a relation modelling ArXiv keys) to FiLiPo. In this way the probing phase will be skipped and the aligning phase starts by using the provided relations. For the same reason we restricted OMDB to using titles only. Furthermore, to use the IMDB dataset we had to set the record overlap threshold from 0.1 to 0.3. This is because IMDB contains a lot of relations with low functionality (e.g. hasLanguage) and therefore incorrect matches would be tolerated.

Since FiLiPo pulls random records from the local KB and uses them to request the API, the alignments found may differ slightly between different runs. The evaluation was therefore performed three times for each combination of KB and API. The average runtime of FiLiPowas around 25 minutes. If the input relations are known, as is the case with DORIS, then the system usually needs no longer than a few minutes because the probing phase can be skipped. The probing phase is expensive in runtime because a significant number of requests are sent to the API (see Table 1). With 25 probing requests as default in the case of the dblp containing 27 possible relations for publications, $25 \times 27 = 675$ requests are made during the probing phase.

FiLiPo was able to determine the correct input relations for all APIs contained in the non-technical set. Error messages such as those returned by SciGraph (a JSON response) were successfully identified in all cases. Thus, the runtime and the number of requests to the API were kept relatively small. For the evaluation we used the metrics precision, recall and F1 Score. FiLiPo was able to achieve a precision between 0.73 to 1.00 and a recall between 0.66 to 1.00. Values close to 1.0 were achieved mainly because there were only a few possible alignments. The corresponding F1 scores for FiLiPo are between 0.69 and 0.95.

Evaluation of DORIS. We re-implemented DORIS as a baseline system for our evaluation. DORIS uses label information of instances as its predefined input relation for APIs. However, this is not always the appropriate input parameter for the API. For example, some APIs expect DOIs as input parameters. In order to extend the set of APIs that can be used with DORIS, we modified DORIS such that the input relation can be specified by the user, allowing to use more sources than its original specification would allow. Since DORIS does not randomly select entities, unlike FiLiPo, there was no need for multiple test runs. DORIS uses two different confidence metrics to determine an alignment: the overlap confidence and PCA confidence. We assessed that the PCA confidence in DORIS is delivers better results for the alignment than the overlap confidence. By using the PCA method DORIS is able to match journal-related relations. Since most of the entities in dblp are conference papers, journal specific relations are lost when using only the overlap confidence. The downside is that a path that was found only once in the API response only needs to match once in order to achieve a high confidence. In such cases it is risky to trust the match and therefore a re-probing is performed. This re-probing increases the runtime considerably, since entities that share the matched relation are subsequently searched and ranked. DORIS has been configured in order to send 100 requests to the APIs. Furthermore, the threshold for the PCA confidence has been set to 0.1 based on a calibration experiment similarly to FiLiPo testing all threshold values between 0.1 and 1.0 (in steps of 0.1). With threshold 0.1, no erroneous alignments were made; recall was significantly larger at 0.1 than with larger values.

FiLiPo outperforms DORIS in terms of precision in most cases and clearly in terms of recall and F1. This is mainly caused by the two disadvantages of DORIS discussed before: First, aligning with entities with most facts often misses rare features of entities (e.g. a specific publisher like Elsevier). As a result, it is not possible for DORIS to determine an alignment between dblp and Elsevier's API. Second, using only one similarity method results in a relatively high precision, but is also too rigid to recognise slightly different data (abbreviations of author names), thus leading to low recall. However, DORIS was able to achieve better results using IMDB, mainly because DORIS excludes all relations with a very low functionality from the alignment process. This way DORIS prevents precision drops but also loses recall as we can see in the other cases.

7 Conclusion

We presented FiLiPo, a system to automatically discover alignments between KBs and APIs. A user only needs knowledge about the KB but no prior knowledge about the API data schema. Our evaluation showed that FiLiPo outperformed DORIS and delivered better results in all but one case. As FiLiPo is currently only able to work with APIs that return a single record, we plan to extend FiLiPo to work with APIs that return more than one response in future work.

References

1. Akbik, A., Blythe, D., Vollgraf, R.: Contextual string embeddings for sequence labeling. In: COLING. pp. 1638–1649 (2018)
2. Bernstein, P.A., Madhavan, J., Rahm, E.: Generic schema matching, ten years later. Proc. VLDB Endow. **4**(11), 695–701 (2011)
3. Dhamankar, R., Lee, Y., Doan, A., Halevy, A.Y., Domingos, P.M.: imap: Discovering complex mappings between database schemas. In: SIGMOD. pp. 383–394. ACM (2004). https://doi.org/10.1145/1007568.1007612
4. Hogan, A., Polleres, A., Umbrich, J., Zimmermann, A.: Some entities are more equal than others: statistical methods to consolidate linked data. In: 4th Workshop on New Forms of Reasoning for the Semantic Web: Scalable & Dynamic (2010)
5. Koutraki, M., Preda, N., Vodislav, D.: SOFYA: semantic on-the-fly relation alignment. In: EDBT. pp. 690–691. OpenProceedings.org (2016). https://doi.org/10.5441/002/edbt.2016.89
6. Koutraki, M., Preda, N., Vodislav, D.: Online relation alignment for linked datasets. In: ESWC. Lecture Notes in Computer Science, vol. 10249, pp. 152–168 (2017). https://doi.org/10.1007/978-3-319-58068-5_10
7. Koutraki, M., Vodislav, D., Preda, N.: Deriving intensional descriptions for web services. In: CIKM. pp. 971–980. ACM (2015). https://doi.org/10.1145/2806416.2806447
8. Koutraki, M., Vodislav, D., Preda, N.: DORIS: discovering ontological relations in services. In: ISWC. CEUR Workshop Proceedings, vol. 1486. CEUR-WS.org (2015)
9. Madhavan, J., Bernstein, P.A., Doan, A., Halevy, A.Y.: Corpus-based schema matching. In: ICDE 2005, 5-8 April 2005, Tokyo, Japan. pp. 57–68. IEEE Computer Society (2005). https://doi.org/10.1109/ICDE.2005.39
10. Madhavan, J., Bernstein, P.A., Rahm, E.: Generic schema matching with cupid. In: VLDB. pp. 49–58. Morgan Kaufmann (2001)
11. Qian, L., Cafarella, M.J., Jagadish, H.V.: Sample-driven schema mapping. In: SIGMOD. pp. 73–84. ACM (2012). https://doi.org/10.1145/2213836.2213846
12. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. VLDB J. **10**(4), 334–350 (2001). https://doi.org/10.1007/s007780100057
13. Suchanek, F.M., Abiteboul, S., Senellart, P.: PARIS: probabilistic alignment of relations, instances, and schema. vol. 5, pp. 157–168 (2011). https://doi.org/10.14778/2078331.2078332