Sample Driven Data Mapping for Linked Data and Web APIs

Tobias Zeimetz
zeimetz@uni-trier.de
Trier University
Trier, Germany

Ralf Schenkel
schenkel@uni-trier.de
Trier University
Trier, Germany

ABSTRACT

In order to create the most comprehensive RDF Knowledge Base possible, data integration is essential. Many different data sources are used to extend a given dataset or to correct errors in the data. Nowadays, Web APIs (instead of data dumps) are common external data sources, since many data providers make their data publicly available. However, the classic problems of data integration, i.e., which parts of the datasets can be mapped, remain. In addition, Web APIs are often more restrictive than data dumps and of course slower to access due to latencies and other constraints. In this paper we demonstrate the FiLiPo (**Fi**nding **Li**nkage **Po**ints) system to automatically find connections (i.e., linkage points) between Web APIs and local Knowledge Bases in a reasonable amount of time. To this end, we developed a sample-driven schema matching system, which models Web API services as parameterized queries. These Web API services return a view definition of their data which subsequently need to be connected to the local database scheme. Furthermore, our approach is able to find valid input values for Web API services automatically (e.g. IDs) and can determine combined linkage points (e.g. first and last name) despite different structures. Our results on six real world API services with two local databases show that our linkage point detection algorithm performs well in terms of precision (0.89 up to 1.0) and recall (0.69 up to 1.0).

CCS CONCEPTS

• **Information systems → Extraction, transformation and loading**; **Mediators and data integration**.

KEYWORDS

Data Integration, Schema Mapping, Relation Alignment

ACM Reference Format:
Tobias Zeimetz and Ralf Schenkel. 2020. Sample Driven Data Mapping for Linked Data and Web APIs. In *submitted to CIKM '20: 29th ACM International Conference on Information and Knowledge Management, October 19–23, 2020, Galway, Ireland.* ACM, New York, NY, USA, 4 pages. https://doi.org/??.????/???????.???????

1 INTRODUCTION

In recent years, RDF Knowledge Bases (KBs) have established themselves as an important database format and are used in many domains. According to Koutraki et al. [4] many of these KBs were generated automatically and are therefore incomplete in their data. Databases generally come with the problem that they are potentially incomplete (considering how much new data is generated daily). For this reason, data integration approaches [3–7] are used to expand KBs and correct erroneous information. As a common data integration process, first external data (usually in form of a data dump) is first downloaded and then aligned with the data contained in a given local KB. "Aligning" describes the process by which relations from the local KB are mapped to relations from external sources, thus creating a mapping between the local and the external data scheme. Using this alignment (also called "mapping"), the actual data integration process can be done and the data of the KB is expanded through missing or new information. For example, in a bibliographic KB like dblp[1] (containing metadata of publications in the domain of computer science), such missing data could be a DOI (Digital Object Identifier) or an author affiliation.

Using APIs for data integration [4, 5] (in contrast to data dumps) has the advantage that the data is usually more up-to-date and due to the possibility of using Web APIs, the number of potential data sources becomes much larger. In addition, most data providers tend to share their data via (RESTful) Web APIs, since this solution seems to be a sweet-spot between making data openly accessible and protecting it [5]. However, the fundamental problems remain the same. The aligning of external data is a difficult task because of the heterogeneity of data structures. In the worst case, the data of the external source is structured completely differently than that of the local KB. For example, in the case of dblp, the author name is modelled as a single property, whilst some external data sources use two properties to model the name, e.g. first and last name. Further problems with matching could also be different spellings, different formatting and so forth. For this reason the data integration process remains a manual task for most parts [5].

Contribution. In this paper we demonstrate FiLiPo (**Fi**nding **Li**nkage **Po**ints)[2], a system to discover alignments between a local KB and an API. In addition, it automatically detects which kind of information has to be sent to an API in order to get a valid response. To this end, we pursue a sample driven approach such as Qian et al. [7] coupled with approaches of DORIS [4, 5]. In the first stage, the system goes through a probing phase, in which various information (e.g. DOIs, titles, etc.) is sent to the API to determine which information the API responds to. Then, the information returned by the API is used to guess its scheme and provide an

[1] https://dblp.uni-trier.de/
[2] https://github.com/dbis-trier-university/FiLiPo

alignment between the local and the external data. In contrast to DORIS we do not use just one string similarity method to determine an alignment rather several different ones. The reason for using more than one similarity method is that a single method is not suited to compare different data types (e.g. names, IDs, etc.). FiLiPo automatically selects the method that performs well for a given data type and determines an alignment. In order to keep the approach simple and usable, even for non-technical users, a user need only specify how many samples should be sent to the Web API.

The *novelty* of the FiLiPo system is that it automatically detects which kind of information has to be sent to the Web API in order to get a valid response. DORIS only uses information associated with the label relation of the local KB and therefore may not use other valid input values, e.g. external IDs that are stored in the KB. The FiLiPo system is, in contrast to DORIS, able to not only find one-to-one matches but also one-to-many matches, i.e. aligning a relation (e.g. full name) with multiple other relations (e.g. first and last name). In contrast to the other systems, instead of using just one similarity method, a group of fifteen similarity methods with multiple variants is used. The best matching similarity method for each relation will be determined automatically and used for the aligning process.

2 SYSTEM OVERVIEW

In this section we describe several use cases for the FiLiPo system and give a brief insight on our user assumptions. Furthermore, we will present a system overview of the FiLiPo system. For the sake of simplicity, the dblp KB will be the standard KB in this paper, but the use cases also apply to KBs of other domains, e.g. cultural heritage, movies and others.

Use Cases. The FiLiPo system tackles the task of finding alignments between a local KB and an API. These alignments can be used for several different use cases. The standard use case is a data integration use case, in which the information of a local KB is extended or missing data is integrated. A classical real world scenario for data integration is dblp. As described in our prior work [9], it is an important task to extend and improve the quality of the information stored in dblp. Therefore, it is necessary to collect and aggregate data from different data repositories such as Springer SciGraph[3] and others. Afterwards, the data can be used in order to improve programs and systems (e.g. recommender systems) that work on the dblp meta data.

Another use case is checking data for correctness. The alignments could be used to compare the data of a local KB with that of multiple APIs. In case a user searches for information about a publication or a conference in dblp, the data of a local KB can be compared with the data of one (or more) APIs. If the data differs too much, the user can be notified that the data may contain errors.

User Assumptions. We assume that the user of the FiLiPo system is a non-technical user without programming knowledge or technical skills. Furthermore, the user has no in-depth knowledge of external data sources, but is familiar with the structure of the local KB. We assume that the user has domain knowledge and therefore can understand common data structures from the genre of the local database (e.g. bibliographic meta data). In addition, an expert with knowledge of the Web APIs (technical user) can make further settings (changing the string similarity thresholds etc.) to fine-tune the system.

[3]https://scigraph.springernature.com/explorer/api/

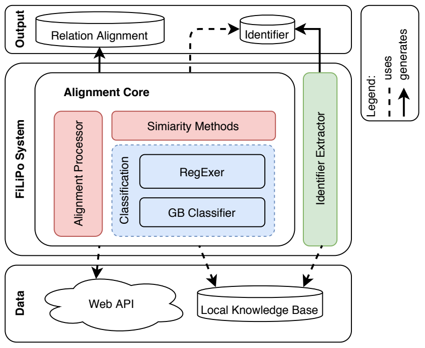

Figure 1: FiLiPo System Architecture

System Architecture. The FiLiPo architecture shown in Figure 1 is divided into two main components, the *Alignment Core* and the *Identifier Extractor*. The Alignment Core consists of three components, namely the *Alignment Processor*, the *Similarity Processor*, the *RegExer*, and the *Classification* component. The Alignment Processor interacts with the other components to determine a correct alignment. The Similarity Processor is used by the Alignment Processor to compare values of a local database and values of an API. It can use 15 different similarity methods with several variants (thus up to 48 methods are available). We used the string similarity library developed by Baltes et al. [2] and used the following three types of similarity methods: (1) equal, (2) edit and (3) set based. String similarity methods of the equal category check for the equality of two strings and edit-based methods define the similarity of two strings based on the number of edit operations (e.g. Levenshtein) needed to transform one string into the other [2]. Set-based methods determine the overlap of two strings in terms of tokens (e.g. n-grams). We excluded the overlap method in the set based category since this method is too fuzzy and would lead to an erroneous aligning (e.g. aligning a title and an abstract since they have in most cases a lot of words/tokens in common).

The Identifier Extractor is used to derive the identifier relations (e.g. ISBNs, ISSNs, etc.) contained in a given local KB. Therefore, the system computes the functionality of each relation. A relation is called functional, if there are no two distinct facts that share the same relation and value [5]. Since real world KBs may be noisy and contain errors, a perfect functional relation is unlikely. Therefore, we used the function presented by Koutraki et al. [8] to compute the functionality score of a relation. In case a relation has a functionality greater or equal to 0.99 (in case the KB contains erroneous data), the system assumes that the relation describes an identifier in the KB. It is important to determine the identifiers of the KB because the FiLiPo system uses several similarity methods that may be too fuzzy to compare identifier values.

The classification component is used by the Alignment Processor in case an identifier value is compared with another potential identifier value. For example, in order to match an ISBN with a value returned by an API the values need to be equal. However, some characters in an ISBN are optional, e.g. hyphens. For this reason using a method of type equals is too strict but using the other similarity methods (e.g Levenshtein) is too fuzzy. To work around these problems, the classification component provides two classifier variants. The first one is a simple regex based variant using predefined rules to compare two values. The second classifier variant is utilising a gradient boosting classifier working on Flair [1] embeddings of identifiers to determine whether two values are equal. We use Flair embeddings instead of other embeddings since this framework is character based and therefore suits better for the comparison of two identifier strings. Both variants can be extended by a technical user by adding rules for the RegExer or by training the gradient boosting classifier for new identifiers. The user can specify in a configuration file whether to use the embedding or regular expression approach.

3 ALIGNING PROCESS

First, the user has to describe every API (URL to the API and input type) that is used by the system in a configuration file. The input type describes the class of entities of the local knowledge base that will be used as input. After starting the system it will ask the user to enter a local KB and an API that will subsequently be aligned. As stated previously, the user additionally needs to specify how many samples are sent to the Web API (sample size).

Probing Phase. After the user has entered all needed information, the system starts the probing phase. It is used to determine what relation of the given input type can be used as valid input value for the used API. To illustrate this with an example, dblp has 29 relations to describe the metadata of a publication (e.g. DOI, ISBN, title, etc.), but Springer's API SciGraph only responds to DOIs. The first step is to send several (initial) requests to the API for each relation of the input type. The number of initial requests is set to 25 for each relation but can be changed via the configuration file by an expert user. However, it turned out that this number of initial requests usually works well, in terms of speed and correctness. The values for the queries are picked uniformly at random from the database in order to cover every version of a type. In the best case scenario, the server of the API responds with the HTTP status code `200 OK` or with an HTTP error code. In the worst case scenario, the server responds all the time with a JSON document containing an error message. In this case the system cannot easily detect that some input values did not lead to a (valid) response and therefore will send further requests to the API in the next step. This would result in a considerable increase of the runtime. It is therefore necessary to identify invalid answers and thus prevent the system from sending additional but unnecessary requests to the API.

In order to identify invalid responses, the system iterates over all answers and compares how similar they are to one another. Invalid answers are similar to one another, since they only contain an error message. In contrast, correct answers are different to one another since they contain information regarding various different entities. As a result, a (potential) error response is determined by counting how often a response was similar (using Levenshtein) to other responses. In this way unnecessary requests to the API are prevented. Finally the matching phase begins by only using the relations that have really led to a valid answer.

Aligning Phase. The first step of the aligning process is to collect more responses from the API for each valid input type (e.g. DOI, ISBN, etc.). In total, as many requests are sent to the API (for each input type) as the user specifies at the beginning (sample size). After collecting additional answers, they are processed by the system. First, path-value pairs are built from the JSON responses to create a structure similar to a graph database (i.e. relation-value-pairs). Then each path-value pair from the response is compared with the relation-value pairs of the corresponding entity from the local database. To determine whether the values of the local and the external record for a relation are the same, all string similarity methods are iterated over and the similarity values are calculated. For this purpose, it is checked whether numbers or URLs are compared. Since string similarity methods are not suitable for these types, they are compared with the *equal* method. Otherwise, all methods are iterated over and the one with the best result is (temporarily) chosen, under the condition that the similarity score is higher than the threshold set by the user (similarity threshold). Afterwards, it is checked whether the relation is an identifier like a DOI. In this case, the similarity is additionally determined with the classifier or the regular expression approach. The threshold is determined prior to this step, because the comparison with the classifier is expensive in time. Thus, only values of the local KB and the API, which are similar enough to be a match, are compared. The second approach to compare identifier values is the regular expression approach. Using predefined rules (which can be extended and changed) various characters are filtered out of the identifier values and compared in the end. This approach is much faster than the gradient boosting approach, because no expensive embedding has to be calculated.

If enough matches are found between the local record and the API record (i.e. data overlap is greater than the overlap threshold specified in the configuration file) then the server response is considered a valid response and is used to determine the final alignment. The data must overlap, because some APIs (similar to retrieval tools) may return an approximately matching response if the data requested is not found.

After collecting all (valid) answers, it is determined which alignments are the most frequent ones. Therefore two different maps are used: Summed Metrics Map (SMM) and Wild Card Map (WCM). In a first step, the SMM is used to summarise all alignments and to determine how often these alignments were previously determined by the program. However, there are alignments that cannot be described with a fix path alone. For example, the title of a paper can always follow the path *record.title*, but the names of the authors vary in their path (e.g. *author[0].name* or *author[1].name*).

In order to determine which path needs to be fix and which needs to be a wild card path the WCM is used. As in the SMM, the first step is to summarise all alignments in the WCM. The only difference in these maps is that the relations of the WCM do not contain fix paths but rather so called branching points (e.g. *author[*].name*). The wild card symbol (*) indicates that more than one branch is possible. For example, to find the missing name of an author, one would have to iterate over the author array and search for the missing name(s).

Using these data structures, the final alignment is calculated. First the combined alignments are determined, then valid wild card alignments, and finally the fix alignments. Combined alignments have to meet several conditions in order to prevent the system from determining erroneous alignments: (1) they have a common prefix (e.g. *record.author[*].given* and *record.author[*].family*), (2) only the suffix may change and (3) they must have been matched similarly often (i.e. a predefined threshold is used). The threshold used for this is predefined in the configuration file and can be changed by experts. However, in our tests we found that the default value of 0.3 worked for all APIs that we tested and gave no incorrect results.

The last step is to determine whether a match between two relations is considered a wild card or a fixed alignment. For each relation from the local KB that is matched with an external relation, the confidence is calculated. This is done only for the wild card alignment that appeared most often. The confidence is calculated by counting each fixed alignment that is equal to the wild card alignment (numbers and wild card symbols are ignored), e.g. *authors[0].fullname* and *authors[*].fullname*. The result is then divided by the number of all fixed alignments that exist for the local relation. If the value is high enough, the wild card alignment is taken as the final match, otherwise the fixed alignment is used. This process is done for each relation of the input type of the local KB and in this way the combined, wild card and fixed alignments form a final alignment between the local KB and the API.

4 DEMONSTRATION

We demonstrate[4] how FiLiPo can assist users in aligning multiple data sources in the context of bibliographic data.

Demonstration Datasets. As local KB we used a dblp RDF dataset and five bibliographic APIs (SciGraph, CrossRef[5], Elsevier[6], ArXiv[7], Semantic Scholar[8]) which are aligned with dblp by the FiLiPo system. All used services respond with metadata about scientific articles and their authors. To test the generic nature of the FiLiPo system we used a KB (Linked Movie DB[9]) and an API (Open Movie Database (OMDB)[10]) from the movie genre.

Demonstration Scenario. We will begin to explain the FiLiPo system by presenting the used datasets, i.e. dblp and CrossRef. We briefly show what kind of data is contained in the sources and what a typical API response looks like. In order to make the problems of aligning clearer to the audience, we will show how different the structure of the data can be and which problems arise from this. Afterwards we will give an overview of the FiLiPoarchitecture as presented in Section 2. Subsequently we will invite the audience to explore the aligning process in a prepared scenario. In the prepared scenario the alignment will only take a short period of time (approximately 2 minutes). During this phase we will briefly explain how the system works and how the final alignment is determined. Furthermore, the regular expression approach is compared with the classifier and the advantages and disadvantages are highlighted. In addition we will show how a technical user can fine-tune the system, e.g. by changing thresholds or the used similarity methods.

Evaluation and Usability. We have evaluated both scenarios (technical user and non-technical user) with the data sets mentioned above. All thresholds were set to default and 100 requests were made. CrossRef, SciGraph, and two services from Semantic Scholar were used for the non-technical scenario and the APIs of OMDB, Elsevier and ArXiv were used to for the technical user scenario. Since Elsevier and the dblp only have a few publications in common, the requests had to be increased to 400. Since ArXiv always returns the top results, except when receiving an ArXiv key we restricted the used relations in the configuration file only to ArXiv keys. For the same reason we restricted OMDB for titles only. The runtime of the system was between 15-32 minutes and increased to 15-37 minutes when the classifier was used. Precision and Recall were determined for evaluation. The precision of the alignments in the tests was between 0.85 and 1.0. The recall took values in the range of 0.67 and 1.0. The system therefore has an F1 score of 0.78 to 1.0.

Related Work. Bernstein et al.[3] briefly summarises some well known techniques and defines a taxonomy for these techniques. One of the mentioned systems, developed by Madhaven et. al., was CUPID [6]. It is used to discover mappings between schema elements based on their names, data types, constraints and schematic structure. Their system consists of two phases: a linguistic matching phase and a structural matching phase.

The system developed by Qian et al. [7] automatically constructs a mapping between the local data and external data (in form of data dumps). The user is not required to have any detailed knowledge about the structures and schemes of any of the data sources. This is an iterative process that produces better mappings, the more sample instances of the local KB the user provides the system with. According to Qian et al. this process lowers the cognitive burden and is therefore also usable for less-technical users.

The closest system to our work is DORIS. The user enters the numbers of requests that will be sent to the API and it will automatically align a local KB with an API. The fundamental idea of DORIS is to use instances of the local KB in order to send sample requests to an API. In order to align the scheme of the external data with the local KB, the values of the relations are compared by using a single string similarity method. All found matches will afterwards be used to align the relations of the data sources.

REFERENCES

[1] A. Akbik, D. Blythe, and R. Vollgraf. 2018. Contextual String Embeddings for Sequence Labeling. In *COLING 2018, 27th International Conference on Computational Linguistics.* 1638–1649.

[2] S. Baltes, L. Dumani, C. Treude, and S. Diehl. 2018. SOTorrent: reconstructing and analyzing the evolution of stack overflow posts. In *MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, A. Zaidman, Z. Kamei, and E. Hill (Eds.). ACM, 319–330.

[3] P. A. Bernstein, J. Madhavan, and E. Rahm. 2011. Generic Schema Matching, Ten Years Later. *PVLDB* 4, 11 (2011), 695–701.

[4] M. Koutraki, N. Preda, and D. Vodislav. 2017. Online Relation Alignment for Linked Datasets. In *ESWC 2017, Portorož, Slovenia, May 28 - June 1, 2017, Proceedings, Part I.* 152–168.

[5] M. Koutraki, D. Vodislav, and N. Preda. 2015. Deriving Intensional Descriptions for Web Services. In *CIKM 2015, Melbourne, VIC, Australia, October 19 - 23, 2015.* 971–980.

[6] J. Madhavan, P. A. Bernstein, and E. Rahm. 2001. Generic Schema Matching with Cupid. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy.* 49–58.

[7] Li Qian, M. J. Cafarella, and H. V. Jagadish. 2012. Sample-driven schema mapping. In *SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012.* 73–84.

[8] F. M. Suchanek, S. Abiteboul, and P. Senellart. 2011. PARIS: Probabilistic Alignment of Relations, Instances, and Schema. *Proc. VLDB Endow.* 5, 3 (2011), 157–168.

[9] T. Zeimetz. 2019. Hybrid Federations - Connecting Web APIs and Linked Data Knowledge Bases. In *GvDB, Saarburg, Germany, June 11-14, 2019. (CEUR Workshop Proceedings),* R. Schenkel (Ed.), Vol. 2367. CEUR-WS.org, 69–73.

[4]Video Link: https://basilika.uni-trier.de/nextcloud/s/vN3Za1gpHmOAEuR
[5]https://www.crossref.org/services/metadata-delivery/rest-api/
[6]https://api.elsevier.com
[7]https://arxiv.org/help/api
[8]https://api.semanticscholar.org
[9]http://www.cs.toronto.edu/~oktie/linkedmdb/linkedmdb-18-05-2009-dump.nt
[10]http://www.omdbapi.com