Incremental Schema Integration for Data Wrangling via Knowledge Graphs

Javier Flores [a], Kashif Rabbani [b], Sergi Nadal [a], Cristina Gómez [a], Oscar Romero [a], Emmanuel Jamin [c] and Stamatia Dasiopoulou [c]

[a] *Department of Service and Information System Engineering, Universitat Politècnica de Catalunya, Barcelona, Spain. E-mails: jflores@essi.upc.edu, snadal@essi.upc.edu, cristina@essi.upc.edu, oromero@essi.upc.edu*

[b] *Department of Computer Science, Aalborg University, Aalborg, Denmark. E-mail: kashifrabbani@cs.aau.dk*

[c] *SEMBU, NTT Data, Barcelona, Spain. E-mails: emmanueljeanjacques.jamin@nttdata.com, stamatia.dasiopoulou@nttdata.com*

Abstract. Virtual data integration is the current approach to go for data wrangling in data-driven decision-making. In this paper, we focus on automating schema integration, which extracts a homogenised representation of the data source schemata and integrates them into a global schema to enable virtual data integration. Schema integration requires a set of well-known constructs: the data source schemata and wrappers, a global integrated schema and the mappings between them. Based on them, virtual data integration systems enable fast and on-demand data exploration via query rewriting. Unfortunately, the generation of such constructs is currently performed in a largely manual manner, hindering its feasibility in real scenarios. This becomes aggravated when dealing with heterogeneous and evolving data sources. To overcome these issues, we propose a fully-fledged semi-automatic and incremental approach grounded on knowledge graphs to generate the required schema integration constructs in four main steps: bootstrapping, schema matching, schema integration, and generation of system-specific constructs. We also present Nextia$_{\text{DI}}$, a tool implementing our approach. Finally, a comprehensive evaluation is presented to scrutinize our approach.

Keywords: Schema integration, Bootstrapping, Virtual data integration

1. Introduction

Big data presents a novel opportunity for data-driven decision-making and modern organizations acknowledge its relevance. Consequently, it is transforming every sector of the global economy and science, and it has been identified as a key factor for growth and well-being in modern societies[1],[2]. With an increasingly large and heterogeneous number of data sources available, it is, however, unclear how to derive value from them [1]. It is, thus, commonplace for any data science project to begin with a data wrangling phase, which entails iteratively exploring heterogeneous data sources to enable exploratory analysis [2]. Let us consider the WISCENTD project[3] of the World Health Organization (WHO) as an exemplary data science project deeply transforming a traditional organization into a data-driven one. The main goal of this project is *to build a system for managing the extraction, storage, and integrated processing of data coming from a variety of data sources related to a group of 21 neglected tropical diseases with diverse and multidimensional natures*. For each of these diseases, the data exchange flows are not well-defined and

[1] https://digital-strategy.ec.europa.eu/en/policies/strategy-data

[2] https://strategy.data.gov/

[3] https://ods.cat/en/who-information-system-to-control-eliminate-ntds-wiscentd/

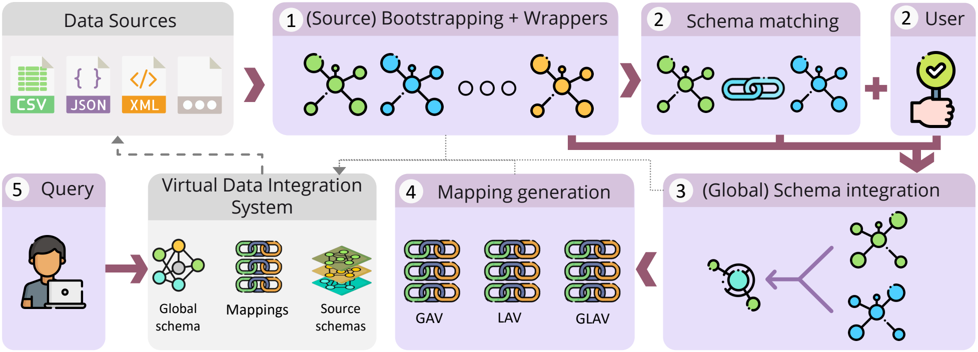

Fig. 1. The schema integration pipeline

drug distributors, pharmacies, health ministries, NGOs, researchers and WHO analysts generate a wealth of data that is ingested into WISCENTD (specifically, in its *Data Lake* platform) with different formats and schema. Most of these data have been generated by third parties and the WHO data scientists are often unaware of their schema, which hinders their capacity to conduct comprehensive data analysis. In this context, data wrangling via virtual data integration is nowadays a popular first step toward understanding the available data sources [3]. Oppositely to physical data integration, where data are warehoused in a fixed target schema, virtual data integration systems keep data in their original sources, build an intermediate infrastructure to provide virtual data integration and provide means to retrieve data at query time [4]. Thus, allowing fast and on-demand data exploration in settings that require fresh data. It is however reported that data scientists spend up to 80% of their time in the manual effort of implementing such data integration pipeline, which is a complex and error-prone process that generates a high-entry barrier [5, 6] for non-IT people. As a result, most current efforts are ad-hoc for a given project and do not generalize. From a theoretical point of view, it is currently the case that the underlying setting has changed (i.e., ill-structured and heterogeneous data sources currently arriving on demand), but the systems supporting data integration are still far from supporting such needs [7]. There is, thus, a growing need for the definition of new approaches that assist and automate such integration pipeline [8].

Data integration encompasses three main tasks: schema integration, record linkage, and data fusion [9, 10]. In this paper, we focus on *schema integration for virtual data integration* to support flexible and on-demand data wrangling. We thus address the problem of automating the extraction of schemata from the sources, as well as their homogenization and integration [11]. The required constructs needed to enable schema integration have been properly studied in the literature [12]: the data source schemata, the target or integrated schema, mappings between the source and target schemata and wrappers fetching the actual data residing in the data sources. Figure 1 exemplifies the traditional steps conducted to build them. The process begins with a (1) bootstrapping and wrapper generation phase, which extracts a schema representation of each source and the program that allows fetching its data. Next, a (2) schema matching phase finds overlapping concepts among different sources, in order to later integrate them. (3) Schema integration is an iterative, user-supervised and incremental task, completed when all source schemata have been integrated into a single and unified global schema. The final phase consists of the (4) generation of mappings, which relate the global schema back to the sources. Virtual data integration systems rely on these constructs to, via rewriting algorithms, automatically rewrite queries over the global schema into queries over the sources [12]. The processes described above do neither support record linkage nor data fusion, which fall out of our scope. Oppositely to a traditional scenario, where all data sources were static, homogeneous and under control of the organization, having a dynamically evolving and heterogeneous set of data sources yields new challenges to efficiently implement such pipeline. We identify the following three:

1. The variety of structured and semi-structured data formats hamper the bootstrapping of the source schemata into a common representation (or canonical data model) that facilitate their homogenization.
2. Instead of the classical waterfall execution, the dynamic nature of data sources introduces the need for a *pay-as-you-go approach* [13] that should support incremental schema integration.

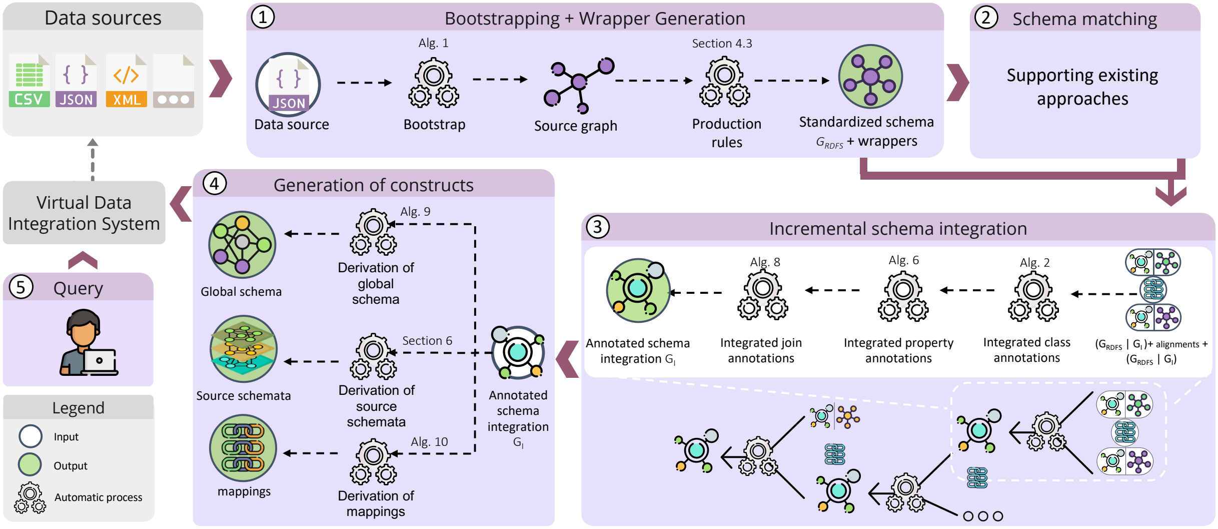
Data sources
CSV
JSON
XML
①
Bootstrapping + Wrapper Generation
Alg. 1
Section 4.3
JSON
Data source
Bootstrap
Source graph
Production rules
Standardized schema
G_{RDFS} + wrappers
②
Schema matching
Supporting existing approaches
Virtual Data Integration System
④
Generation of constructs
Alg. 9
Global schema
Derivation of global schema
Section 6
Source schemata
Derivation of source schemata
Annotated schema integration G_I
Alg. 10
mappings
Derivation of mappings
③
Incremental schema integration
Alg. 8
Alg. 6
Alg. 2
Annotated schema integration G_I
Integrated join annotations
Integrated property annotations
Integrated class annotations
(G_{RDFS} | G_I)+ alignments +
(G_{RDFS} | G_I)
⑤
Query
Legend
Input
Output
Automatic process

Fig. 2. Our approach in a nutshell.

3. The large number and variety of domains in the data sources, some of them unknown at exploration time, hinders the use of predefined domain ontologies to lead the process.

The limitations highlighted above have seriously hampered the development of end-to-end approaches for schema integration, which is a major need in practice, especially when facing Big Data [14, 15]. To that end, we propose an all-encompassing approach that largely automates (essential for Big Data scenarios) schema integration. On top of that, we also implement our approach as an open-source tool called $\text{Nextia}_{\text{DI}}$[4]. Our approach is grounded on knowledge graphs, a widely accepted formalism to represent and homogenize metadata. As depicted in Figure 2, our approach first bootstraps the schemata of structured and semi-structured data formats into source graphs (see phase ①). These are graph-based representations of the particular schema of each data source. Such source graphs are further homogenized into a canonical data model, the RDFS data model, using production rules [16]. Source graphs are accompanied by an automatically generated wrapper that allows to retrieve data from the sources via the source graph. We, next (see phase ②), leverage on the state-of-the-art on schema matching [17] to discover alignments among the source graphs, which are used to guide the incremental schema integration process (see phase ③). There we are able to capture the relationships of the modeled data sources via unions and joins, and generate a graph representing the integration of all data sources' schemata. At each step, our approach extracts and captures rich metadata in the form of RDFS graphs that contain all the required information to generate the schema integration constructs. Such metadata are agnostic of the target system we want to use. Thus, in phase ④, we generate the required schema integration constructs for the target system (e.g., an ontology-based data access system, e.g., [18], or a mediator-based system, e.g., [19]). The system-specific constructs generated are then directly used by the targeted system built-in query rewriting capabilities (as in phase ⑤) to enable on-demand querying and, therefore, data wrangling. The novelty of our approach lies in *the automatic and incremental creation of the constructs required for schema integration of virtual data integration systems*. To our knowledge, this is the only approach supporting this process.

Contributions. We summarize our contributions as follows:

- Our approach is able to deal with heterogeneous and semi-structured data sources.
- It follows an incremental *pay-as-you-go* approach to support end-to-end schema integration.
- It provides a single, uniform and system-agnostic framework grounded on knowledge graphs.

[4] In the ancient Nahuatl language, the term *nextia* means to get something out or put something together

Approach	Required metadata	Supported data model	Implementation	
			Availability	Automation
IncMap [20]	RDB Schema	Relational	Unknown	Semi-automatic
BootOX [21]	RDB Schema	Relational	Unavailable	Automatic
Janus [22] and XS2OWL [23]	XML Schema	XML	Unknown	Automatic
DTD2OWL [24]	DTD Schema	XML	Unknown	Automatic

Table 1

Comparison of bootstrapping state-of-the-art techniques

- We developed and implemented novel algorithms to largely automate all the required phases for schema integration: ① bootstrapping (and wrapper generation) and ③ schema integration.
- Our approach is not specific for a given system and it generates system-agnostic metadata that can be later used to generate the ④ specific constructs of most relevant virtual data integration systems. We showcase it by generating the specific constructs of two representative families of virtual data integration systems (an ontology-based data access system and a mediator-based system).
- We introduce Nextia$_{\mathrm{DI}}$, a tool to support the proposed approach. To our knowledge, Nextia$_{\mathrm{DI}}$ is the very first tool supporting the semi-automatic creation of the schema integration constructs for virtual data integration systems.

Reproducibility and Code Repository. In an attempt to maximize transparency and openness, Nextia$_{\mathrm{DI}}$ is public as an open source Java library, with methods implementing the above mentioned phases. Additionally, the companion website[5] contains details on how to guarantee the reproducibility of the experiments conducted and explained later in the paper to scrutinize our approach.

Outline. The rest of the paper is structured as follows. We first discuss the related work in Section 2. Next, Section 3 presents the formal overview of our approach to further dive in its two main stages: data source bootstrapping (Section 4) and schema integration (Section 5). Section 6 shows how our approach is able to the constructs required by data integration engines. Our approach implemented by Nextia$_{\mathrm{DI}}$ is extensively evaluated in Section 7. We finally conclude our paper and present future work in Section 8.

2. Related Work

Data integration encompasses three main tasks: schema integration, record linkage, and data fusion [9, 10]. A wealth of literature has been produced for each of them. In this paper, we focus on automating and standardizing the process to create the schema integration constructs for virtual data integration systems. Accordingly, in this section, we discuss related work on the two main phases depicted in Figure 2: bootstrapping and schema integration. We wrap up this section discussing the available virtual data integration systems that provide some kind of support to generate the necessary schema integration constructs.

2.1. Related work on bootstrapping

Bootstrapping techniques are responsible for extracting a representation of the source schemata and there has been a significant amount of work in this area, as presented in several surveys [25–28]. Most of the current available efforts, however, do require an a priori available target schema and / or materialize the source data in the global schema [29–31]. Since our approach is bottom-up and meant for virtual systems, we subsequently focus on approaches generating the schema from each source (either checking the available metadata or instances) without using a reference schema or ontology. Table 1 depicts the most representative ones. To better categorize them, we

[5] See more details at https://www.essi.upc.edu/dtim/nextiadi/

Approach	Integration type	Strategy			Implementation	
		Alignments preservation	Source schema preservation	Incremental	Availability	Automation
PROMPT [33]	Full merge				Archieved	Semi-automatic
Chimaera [34] and FCA-Merge [35]	Full merge				Unknown	Semi-automatic
ONION [36]	Simple merge	✓	✓		Unknown	Semi-automatic
OntoMerge [37]	Simple merge	✓	✓		Unavailable	Automatic
ATOM [38]	Asymmetric merge			✓	Unknown	Semi-automatic
CoMerger [39]	Full merge	✓	✓		Active	Automatic
Chevalier et. al [40]	Full merge	✓	✓		Unknown	Semi-automatic

Table 2

Comparison of schema integration state-of-the-art techniques

distinguish three dimensions: required metadata, supported data sources, and implementation, which we detail as follows, accompanied by a discussion to highlight our contributions.

Required metadata. All approaches define transformation rules to generate the schemata from the available data source metadata as in [29], [22], [23], and [24]. However, these transformation rules do require well-defined schemata, even for semi-structured data sources such as XML, which is not realistic for data wrangling tasks. Another relevant problem we identified in all these approaches is the lack of standardization of both the process and the generated schemata. All of them present ad-hoc rules that are not compliant with both the data source metamodel and, importantly, the target metamodel (typically, RDFS or OWL). This hinders the reuse and interoperability of these approaches within the end-to-end schema integration process. Indeed, the same schema processed by different approaches may result in different outputs. This is aligned with the current efforts on schema integration, which are specific for a project and do not generalize well.

Supported data sources. Most approaches focus on bootstrapping relational data sources, assuming a traditional data integration setting, and only three support semi-structured data models [22–24] (in all cases, XML). Yet, in data science, most of the data available comes in the form of semi-structured data sources (e.g., JSON and CSV).

Implementation. Current approaches do not provide an open implementation. BootOX [21] is integrated into Optique system [32], but it is no longer maintained or publicly available.

Discussion. Our bootstrapping proposal differs from the state-of-the-art as follows: (1) the schema is extracted from the physical structure of the available data (i.e., instances) without relying on a pre-defined available schema of the source, (2) the process and produced schema are standardized. Specifically, our transformation rules are defined at the metamodel level and guarantee the output generated is compliant with both the data source and target metamodels. Standardization is key for efficient maintenance and its reuse in the subsequent schema integration phase, and (3) our approach is generic and adaptable to any data source data model. For those models not considered, we could extend our approach by providing a set of transformation rules satisfying the soundness and completeness properties defined in Section 4. Note that these rules are per pair of metamodels and therefore reusable between sources of the same format.

2.2. Related work on schema integration

Schema integration is the process of generating a unified schema that represents a set of source schemata. This process requires as input semantic correspondences between their elements (i.e., alignments). In Table 2, we depict the most relevant state-of-the-art approaches. We distinguish three dimensions: integration type, strategy, and implementation:

Integration type. We follow the classification presented in [41], which categorizes the kinds of integration into three types, namely: simple merge, full merge, and asymmetric merge. The former integrates schemata by adding one-to-one bridge axioms between pairs of schemata. Approaches such as [36] and [37] preserve the original source schemata and alignments, which is a desirable property to manage source evolution [42] and mandatory for incremental approaches. However, their main drawback is the lack of common concepts, which can lead to complex queries when integrating a large number of schemata. Full merge integrates the source schemata generating a new schema where equivalent concepts are merged into a single new concept. In most approaches, the original source schemata is not preserved and therefore lost. Thus, these approaches are not incremental. [39] is the only approach under this category able to preserve specific elements of the source schemas upon request. Asymmetric merge can be performed using the simple and merge schema via one-to-many axioms to integrate schemas and derive a full merge. To the best of our knowledge, [38] is the only approach in this category. This kind of merge prioritizes one of the schemas (target) over the others (source), that is, the target schema will preserve all axioms and in case of disagreement with the target schema, elements of the source schema are removed from the integration.

Incremental integration. Most approaches (e.g., [33, 35, 37]) integrate the source schemata in one shot. Thus, to process a new data source the whole process is started from scratch without reusing the previously generated structures. Instead, [38] reuses the integrated structure to facilitate further integration incrementally. To that end, it adopts the asymmetric merge strategy, however, it only supports the integration of taxonomies (i.e., classes with no attributes and only hierarchical relationships among them).

Implementation. All approaches provide tools, however only [39] is openly available. There exists a demo of [37] on its website[6], however, as of today the service is down. Regarding [33], it is available as a plugin to the Protégé tool, however it is not longer maintained. Overall, we also encountered that none of the tools offer any documentation for users making them hard to use.

Discussion. Our work differs from these works as follows: (1) the global schema is incrementally generated, rather than generated in one shot. (2) We thus incrementally support the integration of any resource (e.g., rdfs:Class and rdf:Property) rather than just focusing in one resource as in [38] (3) In contrast to any other approach, we generate annotations (i.e., metadata) and preserve the original schemata and alignments. These metadata are agnostic of the virtual data integration system at hand, and we are able to derive generic system-specific constructs from our system-agnostic annotations.

2.3. Related work on data integration systems supporting bootstrapping and / or schema integration

We examined virtual data integration systems that have introduced a pipeline supporting bootstrapping and / or schema integration with the goal of providing query access to a set of data sources. We have excluded systems that rely on the manual generation of schema integration constructs [43, 44] and focus on those that support semi-automatic creation of such constructs.In all cases, nevertheless, the processes introduced are specific and dependent on the virtual data integration system at hand. Further, note that some popular data integration systems such as Karma [31], Pool Party Semantic [45], or Ontotext[7] are excluded since they only support physical integration. Table 3 summarizes such efforts.

Optique [32]. It is a virtual data integration system developed as a result of research and industrial projects (e.g., [47]). For each data source, the source schema and RML mappings are generated using BootOX. In order to integrate schemata, equivalent resources between the source and global schemas are mapped manually. During this process, there is no matching technique. Moreover, the global schema is fully maintained by the user, who must update it if it is incomplete with respect to the data sources and propagate the changes throughout the system constructs. However, creating the system constructs does not follow an incremental process. Therefore, new constructs, which are tightly coupled with the system, are needed each time a new data source is added, or the global schema is updated. Finally, Optique allows the integration of relational, streaming, and sensor data and provides a query interface which is performed by the Ontop tool [44].

[6] http://cs-www.cs.yale.edu/homes/dvm/daml/ontology-translation.html

[7] https://www.ontotext.com/

Approach	**end-to-end DI workflow**				**Bootstrapping step**		**Schema integration**		**Implementation**	
	Bootstrapping	**Schema matching**	**Schema integration**	**Query**	**Type**	**Supported data sources**	**Type**	**Incremental**	**Availability**	**Last maintenance**
Optique [32]	✓		✓	✓	Provided and extracted	Relational, sensor data and streaming	Simple merge	-	On request (demo)	2016
Mastro Studio [46]	✓			✓	Provided	Relational	-	-	Commercial use	Unknown

Table 3

Comparison of virtual data integration systems automating, at least partially, schema integration

Mastro studio [46]. It is a virtual data integration system currently available as commercial software maintained by OBDA Systems. The system supports only relational databases, which are mapped to a provided global schema. The mappings are defined in the system's native language, making them tightly coupled to the system. Moreover, there are no schema matching and schema integration phases as it is assumed the user will provide the global schema. Thus, the main complexity is maintaining the global schema and mappings. The query phase explodes the ad-hoc mappings to provide a more efficient query retrieval.

Discussion. All systems generate constructs tightly coupled with the system needs and assume a provided global schema. We, instead, construct the global schema incrementally from the sources, which is crucial for data wrangling. In these approaches, there are three common problems: (1) maintaining the mappings when the schema evolves, (2) updating the global schema to get a complete view of the data sources, and (3) incrementally generating all systems constructs. These problems are even more complex for all the other approaches (e.g., systems and query engines) that fully rely on creating such constructs manually [19, 44, 48–51]. Our approach is the only one incrementally generating and propagating changes. Further, we are not tied to any system. Our approach generates and collects rich metadata from the bootstrapping and schema integration steps. These metadata are rich enough to generate system-specific constructs for schema integration, as we show in Section 6. Our generic metadata is able to capture the evolution and automates its maintenance. Thus, if we need to update the constructs generated for a specific system, we only need to re-run the algorithms to produce the specific constructs and replace them in the system. Moreover, as integration is performed bottom-up, generating a global schema will always be complete with respect to the data sources, which is the main requirement for data wrangling.

3. Approach overview

We now introduce the formal overview of our approach and the running example used in the following sections.

3.1. Formal definitions and approach overview

3.1.1. Data source bootstrapping and production rules

Here, we present the formal definitions that are concerned with phase ① as depicted in Figure 2.

Datasets. Let $\mathbb{M}$ be the set of considered semi-structured and structured data models (e.g., CSV, JSON, or XML), then, a dataset (intuitively, a dataset is each source file) $D_m = \{d_1, \ldots, d_n\}$, where $m \in \mathbb{M}$, is a collection of data elements where each d_i adheres to m's data model. We denote $schema(d)$ the set of elements that represent the physical structure of d's data model (e.g., the set of keys for a JSON document). Since our focus is on data integration, we assume all data elements in a dataset are *homogeneous* (i.e., they share the same schema), formally depicted as $\forall d_i, d_j \in D_m : schema(d_i) = schema(d_j)$, and hence with a slight abuse of notation we will refer to the schema of a dataset as $schema(D_m)$.

Typed graphs. We consider RDF graphs $G = (V_G, E_G)$, which are unweighted, directed edge-labeled graphs. As customary, we see E_G as a set of triples of the form $\langle s, p, o \rangle$, where p is a labeled edge from s to o with $s, o \in V_G$. Then, we say G is typed with respect to a graph $\mathcal{M} = (V_{\mathcal{M}}, E_{\mathcal{M}})$ (i.e., a metamodel), denoted as $G_{\mathcal{M}}$, if for every node $v \in V_G$ there exists a triple in E_G of the form $\langle v, \texttt{rdf:type}, m_i \rangle$, where $m_i \in V_{\mathcal{M}}$ (note we assume that V_G includes the elements of $V_{\mathcal{M}}$). We assume the existence of a set of boolean constraints $\mathcal{P}_{\mathcal{M}}$ for a given metamodel

$\mathcal{M}$, which allow to guarantee that the elements of $G_{\mathcal{M}}$ are well-formed with respect to $\mathcal{M}$. Intuitively, $\mathcal{P}_{\mathcal{M}}$ concerns constraint checking aspects such as generalizations, referential integrity, cardinality, or keys. Then, we say $G_{\mathcal{M}}$ is *consistent* if all constraints in $\mathcal{P}_{\mathcal{M}}$ hold in $G_{\mathcal{M}}$.

Bootstrapping algorithm. A bootstrapping algorithm for a data model m is a function $\mathcal{B}_m : \mathbb{D}_m \rightarrow \mathbb{G}$ from the set $\mathbb{D}_m$ of all datasets adhering to m's data model to the set $\mathbb{G}$ of all graphs. Then, we say $\mathcal{B}_m$ is $\mathcal{M}$*-sound* if the graphs it generates are typed and consistent with respect to a metamodel $\mathcal{M}$. Formally, $\forall D_m \in \mathbb{D}_m$ we have that $\mathcal{B}_m(D_m)$ is typed with respect to $\mathcal{M}$, which we denote $G_{\mathcal{M}} = \mathcal{B}_m(D_m)$.

Graph queries. We consider the query language of *conjunctive graph queries* (CQs) as defined in [52]. Formally, a CQ Q is an expression of the form $Q(x_1, \ldots, x_m) \leftarrow \langle s_1, p_1, o_1 \rangle, \ldots, \langle s_n, p_n, o_n \rangle$, where each s_i, o_j are either vertex labels or variables, and each p_i is either an edge label or a variable. The semantics of answering a query CQ Q over a graph G (i.e., $Q_G(x_1, \ldots, x_m)$) is based on binding nodes and edges from G to the set of variables $\{x_1, \ldots, x_m\}$ that match the conjunction of patterns. In this paper, instead of the traditional homomorphism-based mapping semantics, where different variables can be bound to the same vertex, we assume the more restrictive isomorphism-based semantics, forcing mappings to be injective.

Production rules. Let $G_{\mathcal{S}}, G_{\mathcal{T}}$ be two typed graphs, respectively typed to $\mathcal{S}, \mathcal{T}$. Then, a source-to-target production rule p from $\mathcal{S}$ to $\mathcal{T}$ (i.e., $p_{\mathcal{S} \rightarrow \mathcal{T}}$) is an existential axiom of the form $\forall x \Theta_{G_{\mathcal{S}}}(x) \rightarrow \exists y \Psi_{G_{\mathcal{T}}}(x, y)$, where $\Theta_{G_{\mathcal{S}}}$ is a CQ over $G_{\mathcal{S}}$, and $\Psi_{G_{\mathcal{T}}}$ is a CQs over $G_{\mathcal{T}}$. A production system is a set of production rules $P = \{p_1, \ldots, p_k\}$, with an evaluation function $eval_P : \mathbb{G}_{\mathcal{S}} \rightarrow \mathbb{G}_{\mathcal{T}}$ from the set of all typed graphs with respect to $\mathcal{S}$ (i.e., $\mathbb{G}_{\mathcal{S}}$), to the set of all typed graphs with respect to $\mathcal{T}$ (i.e., $\mathbb{G}_{\mathcal{T}}$). We say a production system is *sound* if, after its evaluation, the resulting graph is typed with respect to $\mathcal{T}$. Formally, $\forall G_{\mathcal{S}} \in \mathbb{G}_{\mathcal{S}}$ we have that $eval_P(G_{\mathcal{S}})$ is typed with respect to $\mathcal{T}$. Likewise, we say a production system is *complete* if, after its evaluation, all candidate nodes in $G_{\mathcal{S}}$ are present in $G_{\mathcal{T}}$.

3.1.2. Integrating bootstrapped graphs and generating the schema integration constructs

Here, we present the formal definitions that are concerned with phases ②, ③ and ④ as depicted in Figure 2.

Alignments. An alignment between two graphs G_A, G_B is a triple of the form $a = \langle v_a, v_b, \ell \rangle$, where v_a and v_b are nodes, respectively in V_{G_A} and V_{G_B}, and ℓ is a user-provided label for the aligned node. An alignment a is $\mathcal{M}$-compliant (i.e., compliant with the metamodel $\mathcal{M}$) if the aligned elements are typed to the same node in $\mathcal{M}$. Formally, $\exists \langle v_a, \texttt{rdf:type}, m \rangle \in E_{G_A}$ and $\exists \langle v_b, \texttt{rdf:type}, m \rangle \in E_{G_B}$, where $m \in V_{\mathcal{M}}$.

Graph integration algorithm. A graph integration algorithm is a function $\mathcal{I} : \overline{\mathcal{A}} \rightarrow \mathcal{G}$ from the set of all sets of alignments to the set of all graphs. We also consider the notion of $\mathcal{M}$-compliant graph integration algorithms, which entails that any graph generated by evaluating $\mathcal{I}$ is typed with respect to $\mathcal{M}$. This is formally defined as $\forall \mathcal{A} \in \overline{\mathcal{A}}$ we have that $\mathcal{I}(\mathcal{A})$ is typed with respect to $\mathcal{M}$.

Schema integration constructs and their generation. We follow Lenzerini's general framework for schema integration[8] [12]. Hence, a schema integration system $\mathcal{K}$ is defined as a triple $\langle \mathcal{G}, \mathcal{S}, \varphi \rangle$, where $\mathcal{G}$ is the global schema, $\mathcal{S}$ the source schemata and φ the mappings between $\mathcal{S}$ and $\mathcal{G}$. Then, following the previous idea, an algorithm to generate the schema integration constructs is a function $\mathcal{Q} : \mathbb{G} \rightarrow \mathbb{K}$ from the set of all graphs to the set of all schema integration systems.

Overview of our approach. Figure 3, summarizes the formal background introduced above and overviews our approach to automate the generation of schema integration constructs from a set of heterogeneous datasets.Shortly, for each dataset D_m compliant with the data model m, we use the bootstrap algorithm $\mathcal{B}_m$ to generate a typed graph of D's schema (i.e., G_m). Such graph is later transformed into another graph G_c, now typed with respect to a canonical metamodel of choice c. Next, and once all graphs have been generated and typed with respect to c, we can execute the graph integration algorithm $\mathcal{I}$ in a pairwise and incremental fashion. Each of the resulting integrated graphs is a candidate to be used to generate the constructs $\mathcal{K}$ for a schema integration system.

[8] Note that virtual data integration systems just consider schema integration and disregard record linkage and data fusion. Thus, in the original framework it talks about data integration, which is used as a synonym of schema integration.

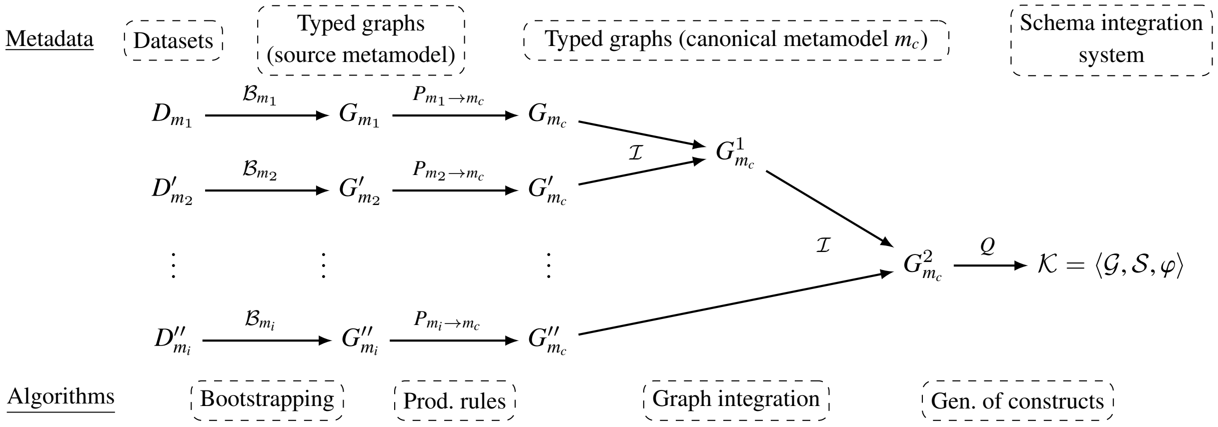

Fig. 3. Formal overview of our approach

3.2. *Running Example*

We consider a data analyst interested in wrangling two different sources about artworks from the Carnegie Museum of Art (CMOA)[9] and Cooper Hewitt Museum (Cooperhewitt)[10]. The former contains information such as the title, creator, and location for all artworks in the museum, such as fine arts, decorative arts, photography, and contemporary art. The latter contains similar information about artworks exposed in the Cooper Hewitt museum such as painted architecture, decorative arts, sculpture and pottery. Figure 4, depicts a fragment of the data in JSON format. Throughout the following sections, we will use this running example to showcase how our approach yields, from these semi-structured data sources, a schema integration system grounded on the previously introduced definitions.

Source: CMOA.json

```
{
   "title": "Keith Haring",
   "web_url": "http://collection.cmoa.org/
↪ CollectionDetail.aspx?item=1",
   "creator": [
      {
         "full_name": "Robert Mapplethorpe",
         "nationality": "American"
      }
   ]
}
```

Source: Cooperhewitt.json

```
{
  "title": "Design for Portion of Stairway",
  "url":
↪ "collection.cooperhewitt.org/objects/68730511/",
  "description": "Drawing, Design for Portion of
↪ Stairway",
  "Participants": [
    {
      "person_name": "Charles Salagnad",
      "person_url":
↪ "collection.cooperhewitt.org/people/18062563/"
    }
  ]
}
```

Fig. 4. Running example excerpts.

4. Data source bootstrapping

As previously described, our approach is generic to any data source, as long as specific algorithms are implemented and shown to satisfy soundness and completeness. In this section, we describe phase ① in Figure 2 and showcase a specific instantiation of the framework for JSON. Following Figure 3, we introduce a bootstrapping algorithm that takes as input a JSON dataset and produces a typed graph-based representation of its schema. Then, such graph is translated into a graph typed with respect to a canonical data model by applying a sound set of production rules. Next, we present the metamodels required for our bootstrapping approach and the production rules.

[9] https://github.com/cmoa/collection
[10] https://github.com/cooperhewitt/collection

4.1. Data source metamodeling

In order to guarantee a standardized process, the first step requires the definition of a metamodel for each data model considered (note that several sources might share the same data model). Figure 5 depicts the metamodel to represent graph-based JSON schemata (i.e., $\mathcal{M}_{\text{JSON}}$). Such metamodel (whose elements we prefix as `J`) describes the basic constructs and the relationships between them. As shown, a J:Document consists of one root J:Object, which in turn contains at least one J:Key instance. Each J:Key is associated with one J:DataType value which is either a J:Primitive, a J:Array or a J:Object. We also assume elements of a J:Array to be homogeneous, and thus it is composed of J:DataType elements. Last, we consider three kinds of primitives: these are J:Number, J:Boolean and J:String. In Appendix A, we present the complete set of constraints that guarantee that any typed graph with respect to $\mathcal{M}_{\text{JSON}}$ is consistent.

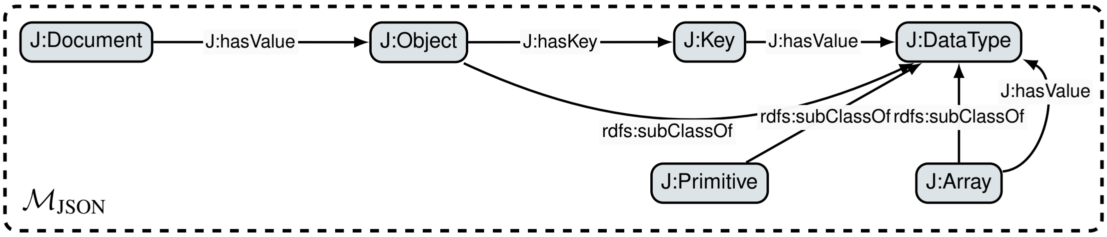

Fig. 5. Metamodel to represent graph-based schemata of JSON datasets (i.e., $\mathcal{M}_{\text{JSON}}$) inspired from [53]

4.2. The canonical metamodel

In order to enable interoperability among graphs typed w.r.t. source-specific metamodels, we choose RDFS as the canonical data model for the integration process. A significant advantage of RDFS is its built-in capabilities for meta-modeling, which supports different abstraction levels. Figure 6, depicts the fragment of the RDFS metamodel ($\mathcal{M}_{\text{RDFS}}$)[11] we adopt. Considering that we aim to represent the schema of the underlying data sources, we only need to instantiate rdfs:Class and their rdf:Property. Additionally, in order to model arrays and under the assumption that there exists a single type of container, we make use of the rdfs:ContainerMembershipProperty property. In Appendix B, we present the complete set of constraints that guarantee that any typed graph with respect to $\mathcal{M}_{\text{RDFS}}$ is consistent.

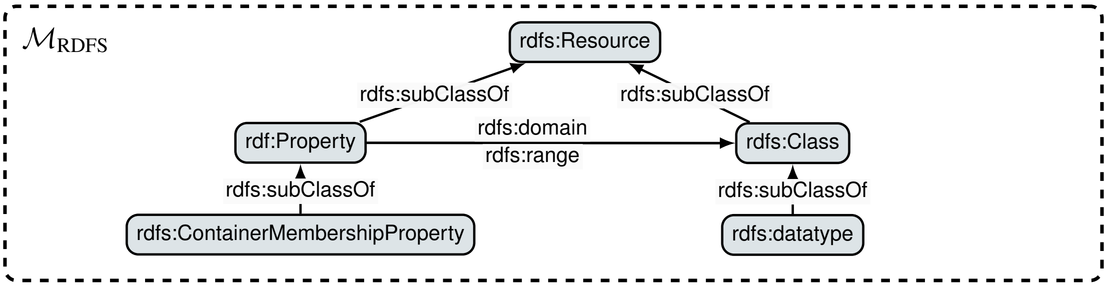

Fig. 6. Fragment of the RDFS metamodel (i.e., $\mathcal{M}_{\text{RDFS}}$) considered in this paper

4.3. Bootstrapping JSON data

We next present a bootstrapping algorithm to generate graph-based representations of JSON datasets. The generation of wrappers that retrieve data using such graph representation has already been studied (e.g., see [19]), and here we reuse such efforts. Here, we focus on the construction of the required data structure. As depicted in Algorithm 1, the method DOCUMENT returns, for a given dataset, a graph typed with respect to the metamodel $\mathcal{M}_{\text{JSON}}$.

[11] https://www.omg.org/spec/ODM/1.1/PDF

It contains a set of functions, one per element of the metamodel, with a shared signature consisting of *a)* the graph G where to populate the triples, which is passed by reference; *b)* the JSON dataset or some of its children (e.g., embedded objects or arrays) D; and *c)* D's parent p (e.g., the key of an embedded object). For simplicity, we make use of a function $\textsc{iri}(s)$, that, given a string s, generates a unique IRI from s. Additionally, we use $\textsc{fresh}()$ to define fresh identifiers (i.e., synthetic strings that do not exist in G), and $\textsc{readFile}(D)$ to read from disk the content of a dataset D.

The goal of this algorithm is to instantiate the J:Document and J:Object elements using the document's root. Then, the method $\textsc{Object}$ recursively instantiates $\mathcal{M}_{\text{JSON}}$ with D's content. Precisely, for each key-value pair $\langle k, v\rangle$, which instantiate J:Key, we define its corresponding J:DataType vt and distinguish the cases of complex (i.e., objects and arrays) and simple (i.e., primitive) elements. Dealing with complex objects requires instantiating J:Object or J:Array with fresh IRIs (i.e., object or array identifiers). This is not the case for primitive elements, which are either connected to the three possible instances of J:Primitive (i.e., J:Number, J:Boolean, or J:String). The presence of such three possible instances in G is guaranteed by function $\textsc{instantiateMetamodel}$. We, additionally, annotate the elements contained in arrays. Assuming that array elements are homogeneous, we relate the instance of J:Array to an instance of J:DataType using the J:hasValue labeled edge.

Algorithm 1 Bootstrap a JSON dataset

```
Input: $D$ is the name/path to the JSON dataset
Output: $G$ is a typed graph with respect to $\mathcal{M}_{\text{JSON}}$ (i.e., $G_{\mathcal{M}_{\text{JSON}}}$)
1: function $\textsc{Document}(D)$
2:     $G \leftarrow \emptyset$
3:     $\textsc{instantiateMetamodel}(G)$
4:     $G \cup= \langle \textsc{iri}(D), \texttt{rdf:type}, \texttt{J:Document} \rangle$
5:     $\textsc{Object}(G, \textsc{readFile}(D), D)$
6:     return $G$

1: function $\textsc{DataType}(G, D, p)$
2:     if $\textsc{isObject}(D)$ then $\textsc{Object}(G, D, p)$
3:     else if $\textsc{isArray}(D)$ then $\textsc{Array}(G, D, p)$
4:     else $\textsc{Primitive}(G, D, p)$

1: function $\textsc{Object}(G, D, p)$
2:     $u' \leftarrow \textsc{fresh}()$
3:     $G \cup= \langle \textsc{iri}(u'), \texttt{rdf:type}, \texttt{J:Object} \rangle$
4:     for all $(k, v) \in D$ do
5:         $G \cup= \langle \textsc{iri}(k), \texttt{rdf:type}, \texttt{J:Key} \rangle$
6:         $G \cup= \langle \textsc{iri}(u'), \texttt{J:hasKey}, \textsc{iri}(k) \rangle\}$
7:         $\textsc{DataType}(G, v, k)$
8:     $G \cup= \langle \textsc{iri}(p), \texttt{J:hasValue}, \textsc{iri}(u') \rangle$

1: function $\textsc{Array}(G, D, p)$
2:     $u' \leftarrow \textsc{fresh}()$
3:     $G \cup= \langle \textsc{iri}(u'), \texttt{rdf:type}, \texttt{J:Array} \rangle$
4:     $\textsc{Datatype}(G, D[0], u')$
5:     $G \cup= \langle \textsc{iri}(p), \texttt{J:hasValue}, \textsc{iri}(u') \rangle$

1: function $\textsc{Primitive}(G, D, p)$
2:     if $\textsc{number}$(D) then $G \cup= \langle \textsc{iri}(p), \texttt{J:hasValue}, \texttt{J:Number} \rangle$
3:     else if $\textsc{bool}$(D) then $G \cup= \langle \textsc{iri}(p), \texttt{J:hasValue}, \texttt{J:Boolean} \rangle$
4:     else $G \cup= \langle \textsc{iri}(p), \texttt{J:hasValue}, \texttt{J:String} \rangle$

1: function $\textsc{instantiateMetamodel}(G)$
2:     $G \cup= \mathcal{M}_{\text{JSON}}$
3:     $G \cup= \langle \texttt{J:Number}, \texttt{rdf:type}, \texttt{J:Primitive} \rangle$
4:     $G \cup= \langle \texttt{J:Boolean}, \texttt{rdf:type}, \texttt{J:Primitive} \rangle$
5:     $G \cup= \langle \texttt{J:String}, \texttt{rdf:type}, \texttt{J:Primitive} \rangle$
```

$\textsc{Document}(\mathit{CMOA.json})$	$\langle \mathit{CMOA.json}, \texttt{rdf:type}, \texttt{J:Document} \rangle$
$\textsc{instantiateMetamodel}(G)$	$\langle \texttt{J:String}, \texttt{rdf:type}, \texttt{J:Primitive} \rangle$ $\langle \texttt{J:Number}, \texttt{rdf:type}, \texttt{J:Primitive} \rangle$ $\langle \texttt{J:Boolean}, \texttt{rdf:type}, \texttt{J:Primitive} \rangle$
$\textsc{Object}(G, \{\ldots\}, \mathit{CMOA.json})$	$\langle o_1, \texttt{rdf:type}, \texttt{J:Object} \rangle$ $\langle \mathit{title}, \texttt{rdf:type}, \texttt{J:Key} \rangle$ $\langle o_1, \texttt{J:hasKey}, \mathit{title} \rangle$ $\langle \mathit{creator}, \texttt{rdf:type}, \texttt{J:Key} \rangle$ $\langle o_1, \texttt{J:hasKey}, \mathit{creator} \rangle$ $\langle \mathit{CMOA.json}, \texttt{J:hasValue}, o_1 \rangle$
$\textsc{Primitive}(G, \texttt{String}, \mathit{title})$	$\langle \mathit{title}, \texttt{J:hasValue}, \texttt{J:String} \rangle$
$\textsc{Array}(G, [\ldots], \mathit{creator})$	$\langle a_1, \texttt{rdf:type}, \texttt{J:Array} \rangle$ $\langle \mathit{creator}, \texttt{J:hasValue}, a_1 \rangle$
$\textsc{Object}(G, \{\ldots\}, a_1)$	$\langle o_2, \texttt{rdf:type}, \texttt{J:Object} \rangle$ $\langle \mathit{full\_name}, \texttt{rdf:type}, \texttt{J:Key} \rangle$ $\langle o_2, \texttt{J:hasKey}, \mathit{full\_name} \rangle$ $\langle a_1, \texttt{J:hasValue}, o_2 \rangle$
$\textsc{Primitive}(G, \texttt{String}, \mathit{f\_name})$	$\langle \mathit{full\_name}, \texttt{J:hasValue}, \texttt{J:String} \rangle$

Fig. 7. Set of triples generated by Algorithm 1. In the left-hand side we depict the function that generates each corresponding set of triples

Example 1. *Retaking the running example introduced in Section 3.2, Figure 7 depicts the set of triples that are generated by Algorithm 1 on a simplified version of the CMOA.json dataset. Note that, for the sake of simplicity, the keys* `web_url` *and* `nationality` *have been omitted.*

Proof of soundness. We show that Algorithm 1 is $\mathcal{M}_{\text{JSON}}$-sound. This is, for any input JSON dataset, the graph it generates is typed with respect to $\mathcal{M}_{\text{JSON}}$ and consistent with respect to the set of boolean constraints $\mathcal{P}_{\mathcal{M}_{\text{JSON}}}$. By construction we can see that for every node $v \in V_G$ (where G is the output of Algorithm 1) there exists an edge $\langle v, \texttt{rdf:type}, m_i \rangle$ where $m_i \in V_{\mathcal{M}_{\text{JSON}}}$. Indeed, all instances of J:Document are declared in line 4 of function $\textsc{Document}$, while all instances of J:Object and J:Key are declared, respectively, in lines 3 and 5 of function $\textsc{Object}$.

Instances of J:Array are declared in line 3 of function ARRAY. Regarding primitive data types, they are all declared once when instantiating the metamodel.

Proof of completeness. We show that Algorithm 1 is complete with respect to the JSON dataset. That is, for any JSON key k, there exists a resource R in the generated graph. This is guaranteed since for every element in $\mathcal{M}_{\text{JSON}}$ there exists a function generating a resource R in the graph. Indeed, for J:Document, J:Object, J:Primitive, J:Array and J:DataType there exists a method named likewise. Regarding J:Key, the resource R is created in the method OBJECT. Note that we assume keys in a JSON dataset contain schema values and not data values (i.e., they are well-formed). We also assume that all elements in an array are homogeneous.

4.4. Production rules

We next present the second step of our bootstrapping approach (see Fig. 3), which is the translation of graphs typed with respect to a specific source data model (e.g., $\mathcal{M}_{\text{JSON}}$) into graphs typed with respect to a canonical metamodel (i.e., $\mathcal{M}_{\text{RDFS}}$). Following the idea in [16], we present the production system used to translate graphs typed with respect to JSON to RDFS. The production system $P_{\text{JSON}\rightarrow\text{RDFS}}$ consists of five production rules which can be evaluated in no particular order. The remainder of this subsection is devoted to formally present the production rules that compose $P_{\text{JSON}\rightarrow\text{RDFS}}$. It is worth noting the usage of the Kleene closure operator in Rules #4 and #5, which allows to represent multidimensional arrays in the generated graph.

Rule 1. Instances of J:Object are translated to instances of rdfs:Class.

$$\forall o\big(\langle o, \texttt{rdf:type}, \texttt{J:Object}\rangle(G)\big) \implies \exists c\big(\langle c, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle(G') \land c = o\big) \tag{R_1}$$

Rule 2. Instances of J:Key are translated to instances of rdf:Property. Additionally, this requires defining the rdfs:domain of such newly defined instance of rdf:Property.

$$\forall o, k\big(\langle o, \texttt{J:hasKey}, k\rangle(G)\big) \implies \begin{array}{l} \exists c, p(\langle p, \texttt{rdf:type}, \texttt{rdf:Property}\rangle(G') \land \\ \langle p, \texttt{rdfs:domain}, c\rangle(G') \land p = k \land c = o) \end{array} \tag{R_2}$$

Rule 3. Instances of J:Key which have a value an instance of J:Array are also instance of rdfs:ContainerMembershipProperty.

$$\begin{array}{l} \forall o, k\big(\langle o, \texttt{J:hasKey}, k\rangle(G) \land \\ \langle k, \texttt{J:hasValue}, a\rangle(G) \land \\ \langle a, \texttt{rdf:type}, \texttt{J:Array}\rangle(G)\big) \end{array} \implies \exists p(\langle p, \texttt{rdf:type}, \texttt{rdfs:ContMembProperty}\rangle(G') \land p = k) \tag{R_3}$$

Rule 4. The rdfs:range of an instance of J:Primitive is its corresponding counterpart in the `xsd` vocabulary. Below we show the case for instances of J:String whose counterpart is xsd:string. The procedure for instances of J:Number and J:Boolean is similar using their pertaining type.

$$\begin{array}{l} \forall k, v\big(\langle k, \texttt{J:hasValue+}, v\rangle(G) \land \\ \langle k, \texttt{rdf:type}, \texttt{J:Key}\rangle(G) \land \\ \langle v, \texttt{rdf:type}, \texttt{J:String}\rangle(G)\big) \end{array} \implies \begin{array}{l} \exists p\big(\langle p, \texttt{rdf:type}, \texttt{rdf:Property}\rangle(G') \land \\ \langle p, \texttt{rdfs:range}, \texttt{xsd:string}\rangle(G') \land p = k\big) \end{array} \tag{R_4}$$

Rule 5. The rdfs:range of an instance of J:Object is the value itself.

$$\begin{array}{l} \forall k, v\big(\langle k, \texttt{J:hasValue+}, v\rangle(G) \land \\ \langle k, \texttt{rdf:type}, \texttt{J:Key}\rangle(G) \land \\ \langle v, \texttt{rdf:type}, \texttt{J:Object}\rangle(G)\big) \end{array} \implies \exists p, r\big(\langle p, \texttt{rdfs:range}, r\rangle(G') \land p = k \land r = v\big) \tag{R_5}$$

Example 2. *Here, we take as input the graph generated by our bootstrapping algorithm depicted in Figure 7. Then, Figure 8 shows the resulting graph typed w.r.t. the RDFs metamodel after applying the production rules. Each node and edge is annotated with the rule index that produced it.*

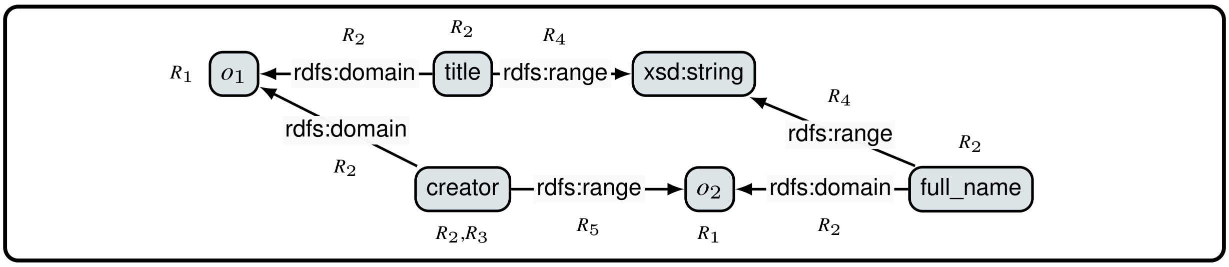

Fig. 8. Graph typed w.r.t. the RDFS metamodel resulting from evaluating the production rules

5. Incremental schema integration

This section describes phase ③ in Figure 2, which addresses the problem of generating the *integrated graph* (IG) from a pair of typed graphs, as illustrated in Figure 3. To facilitate the schema integration process, we extend the fragment of the RDFS metamodel (See Section 4.2) to include two new resources, namely :IntegratedResource and :JoinProperty (see Figure 9). These resources are essential to annotate how the underlying data sources must be integrated (e.g., union or join). The integration is accomplished through a set of invariants (see Section 5.1) that guarantee the completeness and incrementality of the IG. Importantly, our proposal is sound and complete with regard to the bootstrapping method described in Section 4 and it can only guarantee these properties if its input is typed graphs (see Figure 3). The algorithm we propose in this phase takes as input two typed graphs and a list of semantic correspondences (i.e., alignments) between them. The integration algorithm is guided by semantic correspondences when generating the IG, keeping track of used and unused alignments in the process. Unused alignments identify those semantic correspondences not yet integrated due to conditions imposed by the integration algorithm, but to be integrated once the conditions are met in further executions. To obtain the semantic correspondences, we rely on existing schema matching techniques (e.g., LogMap [54] or Nextia$_{\text{JD}}$[17]) to produce alignments.

All in all, the IG is a rich set of metadata containing all relevant information to perform schema integration and, as such, it traces all information from the sources as well as that of integrated resources to support their incremental construction. In Appendix B.1, we present the complete set of constraints that guarantee that any IG is consistent.

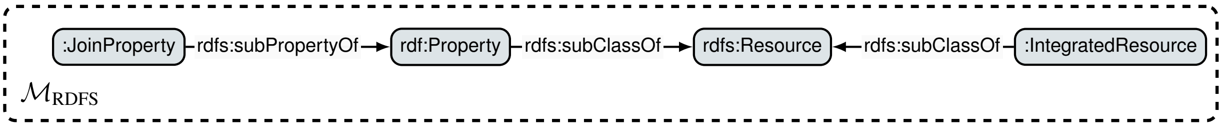

Fig. 9. Extension of the rdfs metamodel for the integration annotation

Example 3. *Continuing the running example, Figure 10 illustrates the result of the bootstrapping step to generate the typed graph representation of the data sources CMOA.json and Cooperhewitt.json. Note that the range properties are omitted due to space reasons. Also, we distinguish* Datatype property *and* Object property*. Since this information is not available in the input typed graphs, we use* rdf:Property *and distinguish them by checking their range. Hereinafter, as shown in Figure 10, we use the following color schemes to represent* rdfs:Class*,* Datatype property *and* Object property*. We use dark colors to represent integrated resources:* :IntegratedResource *of type class,* :IntegratedResource *of type datatype property and* :IntegratedResource *of type object property.*

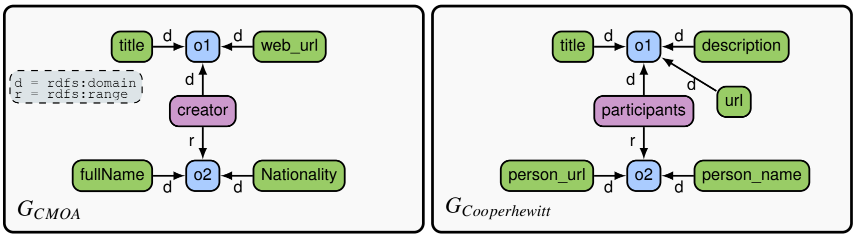

Fig. 10. The extracted canonical RDF representations for the CMOA and Cooperhewitt sources

5.1. *Integration of resources*

We here present an implementation for the graph integration algorithm (see Section 3.1.2). This algorithm, system-agnostic (i.e., not tied to any specific system) and incremental by definition, generates an integrated graph G_I for the input typed graphs. Our algorithm requires as input two typed graphs G_A and G_B and a list of alignments A (see Section 3 for a precise definition of the typed graph and the kind of alignments $\langle R_A, R_B, l\rangle$ considered). Intuitively, we generate G_I as the union of G_A and G_B plus a set of integrated resources generated from the input alignments. As shown in Figure 3, the integration is performed pairwise and the result of executing our algorithm is always an IG. Importantly, an IG is, by definition, a typed graph and therefore it guarantees the *closure* of our algorithm: i.e., it can take as input a previously generated G_I and a typed graph produced from a new data source and generate a new IG, G'_I, that integrates both inputs. The integration of N data sources requires the execution of this algorithm $N-1$ times and generates $N-2$ intermediate IGs prior generating the final IG integrating all sources. In the following, we present the invariants that guarantee the creation and propagation of the integrated resources into the final IG, which hereinafter we will refer to as G'_I. We first present the invariants for class resources.

I1. If R_A and R_B are not :IntegratedResource, then R_A and R_B become subclass of a new :IntegratedResource R_l defined as the URI representation of l (i.e., $R_l = generateURI(l)$). Formally:

$$\begin{array}{l}
\forall\langle R_A, R_B, l\rangle \in A : \\
\langle R_A, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_I \wedge \\
\langle R_B, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_I \wedge \\
\langle R_A, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_I \wedge \\
\langle R_B, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_I
\end{array}
\implies
\begin{array}{l}
\exists R_l = generateURI(l) : \\
\langle R_l, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \in G_{I'} \wedge \\
\langle R_l, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_{I'} \wedge \\
\langle R_A, \texttt{rdfs:subClassOf}, R_l\rangle \in G_{I'} \wedge \\
\langle R_B, \texttt{rdfs:subClassOf}, R_l\rangle \in G_{I'}
\end{array}$$

I2. If R_A is not an :IntegratedResource and R_B is an :IntegratedResource, then R_A becomes a subclass of R_B. Formally:

$$\begin{array}{l}
\forall\langle R_A, R_B, l\rangle \in A : \\
\langle R_A, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_I \wedge \\
\langle R_B, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \in G_I \wedge \\
\langle R_A, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_I \wedge \\
\langle R_B, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_I
\end{array}
\implies
\begin{array}{l}
\langle R_A, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_{I'} \wedge \\
\langle R_B, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \in G_{I'} \wedge \\
\langle R_B, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_{I'} \wedge \\
\langle R_A, \texttt{rdfs:subClassOf}, R_B\rangle \in G_{I'}
\end{array}$$

I3. If R_A is an :IntegratedResource and R_B is not an :IntegratedResource, then R_B is a subclass of R_A. Formally:

$$\begin{array}{l}
\forall\langle R_A, R_B, l\rangle \in A : \\
\langle R_A, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \in G_I \wedge \\
\langle R_B, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_I \wedge \\
\langle R_A, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_I \wedge \\
\langle R_B, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_I
\end{array}
\implies
\begin{array}{l}
\langle R_A, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \in G_{I'} \wedge \\
\langle R_A, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle \in G_{I'} \wedge \\
\langle R_B, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_{I'} \wedge \\
\langle R_B, \texttt{rdfs:subClassOf}, R_A\rangle \in G_{I'}
\end{array}$$

I4. If R_A and R_B are :IntegratedResource, then the new integrated resource C_l defined as the URI representation of l (i.e., $C_l = generateURI(l)$) replaces R_A and R_B. Formally:

$$\begin{array}{l} \forall \langle R_A, R_B, l\rangle \in A: \\ \langle R_A, \mathtt{rdf{:}type}, \mathtt{{:}IntegratedResource}\rangle \in G_I \wedge \\ \langle R_B, \mathtt{rdf{:}type}, \mathtt{{:}IntegratedResource}\rangle \in G_I \wedge \\ \langle R_A, \mathtt{rdf{:}type}, \mathtt{rdfs{:}Class}\rangle \in G_I \wedge \\ \langle R_B, \mathtt{rdf{:}type}, \mathtt{rdfs{:}Class}\rangle \in G_I \wedge \\ \langle r, \mathtt{rdfs{:}subClassOf}, R_A\rangle \in G_I \wedge \\ \langle r', \mathtt{rdfs{:}subClassOf}, R_B\rangle \in G_I \wedge r \neq r' \end{array} \Longrightarrow \begin{array}{l} \exists R_l = generateURI(l) \wedge \forall r, r': \\ \langle R_l, \mathtt{rdf{:}type}, \mathtt{{:}IntegratedResource}\rangle \in G_{I'} \wedge \\ \langle R_l, \mathtt{rdf{:}type}, \mathtt{rdfs{:}Class}\rangle \in G_{I'} \wedge \\ \langle R_A, \mathtt{rdf{:}type}, \mathtt{{:}IntegratedResource}\rangle \notin G_{I'} \wedge \\ \langle R_A, \mathtt{rdf{:}type}, \mathtt{rdfs{:}Class}\rangle \notin G_{I'} \wedge \\ \langle R_B, \mathtt{rdf{:}type}, \mathtt{{:}IntegratedResource}\rangle \notin G_{I'} \wedge \\ \langle R_B, \mathtt{rdf{:}type}, \mathtt{rdfs{:}Class}\rangle \notin G_{I'} \wedge \\ \langle r, \mathtt{rdfs{:}subClassOf}, R_A\rangle \notin G_{I'} \wedge \\ \langle r', \mathtt{rdfs{:}subClassOf}, R_B\rangle \notin G_{I'} \wedge \\ \langle r, \mathtt{rdfs{:}subClassOf}, C_l\rangle \in G_{I'} \wedge \\ \langle r', \mathtt{rdfs{:}subClassOf}, C_l\rangle \in G_{I'} \end{array}$$

Intuitively, the presented invariants guarantee the soundness and completeness of our approach, since it covers all potential cases derived from one-to-one mappings: non-previously integrated resources, either one or another previously integrated or both already integrated. Grounded on these invariants, algorithm 2 generates the corresponding integrated metadata and supports their incremental construction and propagation. Note that invariant *I1* creates a new :IntegratedResource of type class, invariants *I2* and *I3* reuse an existing :IntegratedResource and invariant *I4* replace an :IntegratedResource. To accomplish *I4*, our algorithm uses the method *replaceIntegratedResource* depicted in Algorithm 3, which provides the necessary functionality to replace the resources that were integrated by an :IntegratedResource$_{old}$ with :IntegratedResource$_{new}$. When integrating classes, all resources (e.g., properties) connected to :IntegratedResource$_{old}$ must be connected to :IntegratedResource$_{new}$ For example, replacing an :IntegratedResource$_{old}$ of type class requires updating all properties referencing :IntegratedResource$_{old}$ by rdfs:range or rdfs:domain to reference :IntegratedResource$_{new}$. Therefore, we introduce the method *concordanceRelations* to redirect the relations from an old resource to a new resource.

Algorithm 2 Integration of resources – Classes

Input: Two typed graphs G_A and G_B with a set of class alignments $A = \{a_1, ..., a_n\}$ and a set of unused alignments A_{unused} from previous integrations (For the first iteration, this parameter is an empty set)
Output: Generates the integrated graph $G_{I'}$ and the set of alignments not used $A_{unused'}$

```
function IntegrateResources($G_A$, $G_B$, $A$, $A_{unused}$)
    $G_I \leftarrow G_A \cup G_B$
    $G_{I'} \leftarrow G_I$
    $A_{unused'} = A_{unused}$
    for all $\langle R_A, R_B, l\rangle$ in $A$ do
        $R_l \leftarrow generateURI(l)$
        if any condition then
            if $\langle R_A, \mathtt{rdf{:}type}, \mathtt{{:}IntegratedResource}\rangle \in G_{I'} \wedge \langle R_B, \mathtt{rdf{:}type}, \mathtt{{:}IntegratedResource}\rangle \in G_{I'}$ then    ▷ Invariant I4
                $G_{I'}$ \= $replaceIntegratedResource(G_{I'}, R_A, R_l)$    ▷ see Algorithm 3
                $G_{I'}$ \= $replaceIntegratedResource(G_{I'}, R_B, R_l)$
            else if $\langle R_A, \mathtt{rdf{:}type}, \mathtt{{:}IntegratedResource}\rangle \in G_{I'}$ then    ▷ Invariant I3
                $G_{I'} \cup= \langle R_B, \mathtt{rdfs{:}subClassOf}, R_A\rangle$
            else if $\langle R_B, \mathtt{rdf{:}type}, \mathtt{{:}IntegratedResource}\rangle \in G_{I'}$ then    ▷ Invariant I2
                $G_{I'} \cup= \langle R_A, \mathtt{rdfs{:}subClassOf}, R_B\rangle$
            else    ▷ Invariant I1
                $G_{I'} \cup= \langle R_l, \mathtt{rdf{:}type}, \mathtt{{:}IntegratedResource}\rangle$
                $G_{I'} \cup= \langle R_l, \mathtt{rdf{:}type}, \mathtt{rdfs{:}Class}\rangle$
                $G_{I'} \cup= \langle R_A, \mathtt{rdfs{:}subClassOf}, R_l\rangle$
                $G_{I'} \cup= \langle R_B, \mathtt{rdfs{:}subClassOf}, R_l\rangle$
        else
            $A_{unused'} \cup= (R_A, R_B, l)$
    return $\langle G_I, A_{unused'}\rangle$
```

Algorithm 4 is the main integration algorithm to generate G'_I. It expects sets of alignments to integrate resources pertaining to the same type and implements the methods *IntegrateClasses* for class resources, *IntegrateDataTypeProperties* for datatype properties and *IntegrateObjectProperties* for object properties following the rationale previously presented and guaranteeing the class and property invariants.

Algorithm 3 Replace integrated resource – Classes

```
Input: $G_I$ is the integrated graph, $R_{old}$ is the old integrated resource to be replaced , $R_{new}$ is the new integrated resource
Output: Generates the integrated graph $G_{I'}$ containing the new triples with $R_{new}$

function ReplaceIntegratedResource($G_I, R_{old}, R_{new}$)
    $G_{I'} \leftarrow G_I$
    $G_{I'} \cup= \langle R_{new}, \texttt{rdf:type}, \texttt{:IntegratedResource} \rangle$
    $G_{I'} \setminus= \langle R_{old}, \texttt{rdf:type}, \texttt{:IntegratedResource} \rangle$
    for all $c$ in $\langle ?r, \texttt{rdfs:subClassOf}, R_{old} \rangle (G_{I'})$ do
        $G_{I'} \setminus= \langle r, \texttt{rdfs:subClassOf}, R_{old} \rangle$
        $G_{I'} \cup= \langle r, \texttt{rdfs:subClassOf}, R_{new} \rangle$
        $G_{I'} \cup= concordanceRelations(G_{I'}, R_{old}, R_{new})$
    return $G_{I'}$
```

Algorithm 4 Incremental schema integration

```
Input: Two typed graphs $G_A$ and $G_B$ with a set of class alignments $A_C$, data type property alignments $A_{DTP}$, object property alignments $A_{OP}$ and any property alignments $A_{unused}$ not used in previous integrations (For the first iteration, this parameter is an empty set)
Output: Generates the integrated graph $G_I$ and a set of property alignments $A_{unused'}$

function $\mathcal{I}(G_A, G_B, A_C, A_{DTP}, A_{OP}, A_{unused})$
    $G_I \leftarrow G_A \cup G_B$
    $\langle G_I, A_{unused'} \rangle = IntegrateClasses(G_I, A_C, A_{unused})$
    $\langle G_I, A_{unused'} \rangle = IntegrateDataTypeProperties(G_I, A_{DTP}, A_{unused'})$
    $\langle G_I, A_{unused'} \rangle = IntegrateObjectProperties(G_I, A_{OP}, A_{unused'})$
    return $\langle G_I, A_{unused'} \rangle$
```

Algorithm 5 ConcordanceRelations – Classes

```
Input: $G_I$ is the integrated graph, $C_{old}$ is the old integrated class to be replaced, $C_{new}$ is the new integrated class
Output: Generates the integrated graph $G_{I'}$ containing the new triples with $C_{new}$

function ConcordanceRelations($G_I, C_{old}, C_{new}$)
    for all $p$ in $\langle ?p, \texttt{rdfs:domain}, C_{old} \rangle (G_{I'})$ do
        $G_{I'} \setminus= \langle p, \texttt{rdfs:domain}, C_{old} \rangle$
        $G_{I'} \cup= \langle p, \texttt{rdfs:domain}, C_{new} \rangle$
    for all $p$ in $\langle ?p, \texttt{rdfs:range}, C_{old} \rangle (G_{I'})$ do
        $G_{I'} \setminus= \langle p, \texttt{rdfs:range}, C_{old} \rangle$
        $G_{I'} \cup= \langle p, \texttt{rdfs:range}, C_{new} \rangle$
    return $G_{I'}$
```

G_{CMOA}	$G_{Cooperhewitt}$	user-provided label l
o1	o1	Artworks
title	title	Title
web_url	url	URL
o2	o2	Creator
fullName	personName	Name
creator	participants	Contributors

Table 4

Discovered correspondences for G_{CMOA} and $G_{Cooperhewitt}$

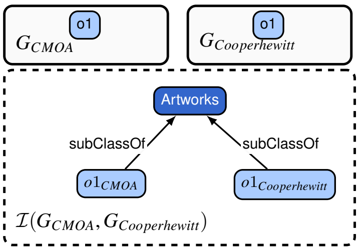

Fig. 11. Integration of two classes from G_{CMOA} and $G_{Cooperhewitt}$

5.1.1. *Integration of classes*

We have introduced the *IntegrateClasses* method implementing the class invariants presented in Algorithm 2. Note that, for invariant 4, Algorithm 5 illustrates the method *concordanceRelations* to ensure that all rdf:Property axioms are coherent when replacing an :IntegratedResource.

Example 4. *Retaking the running example, consider the alignments depicted in Table 4 between G_{CMOA} and $G_{Cooperhewitt}$. Let us consider the alignment* o1 *and* o1*. For this case, invariant I1 is applied as illustrated in Figure 11. Thus, both classes are connected to a newly defined instance of class* :IntegratedResource *and* rdfs:Class*, namely* Artworks*. For this example, two* :IntegratedResource *were generated:* Artworks *and* Creator*. Note, the set A_{unused} is empty since this is the first integration performed.*

5.1.2. *Integration of properties*

The invariants for properties are very similar to those presented for classes. Thus, we will use the *IntegrateClasses* method as a baseline to discuss the integration of properties. Importantly, we follow a class-oriented integration for properties, that is, properties can only be integrated once their corresponding domain and range classes have

Algorithm 6 Integration of Data type properties — adapted from Algorithm 2

```
7:  $C_{domainA} \leftarrow GetIntegratedClass(G_{I'}, R_A, \texttt{rdfs:domain})$
8:  $C_{domainB} \leftarrow GetIntegratedClass(G_{I'}, R_B, \texttt{rdfs:domain})$
9:  if $\langle C_{domainA}, \texttt{rdf:type}, :IntegrationClass\rangle \in G_{I'} \wedge C_{domainA} = C_{domainB}$ then
10:     ...                                          ▷ Invariants I4, I3, I2
16:     ...                                          ▷ Start of Invariant I1
17:     $C_{rangeA} \leftarrow GetIntegratedClass(G_{I'}, R_A, \texttt{rdfs:range})$
18:     $G_{I'} \cup= \langle R_l, \texttt{rdfs:range}, C_{rangeA}\rangle$
19:     $G_{I'} \cup= \langle R_l, \texttt{rdfs:domain}, C_{domainA}\rangle$
20:     ...                                          ▷ End of Invariant I1
21: else
22:     $A_{unused'} \cup= (R_A, R_B, l)$
23: return $\langle G_{I'}, A_{unused'}\rangle$
```

Algorithm 7 ConcordanceRelations — Properties

Input: G_I is the integrated graph, P_{old} is the old integrated property to be replaced, P_{new} is the new integrated property
Output: Generates the integrated graph $G_{I'}$ containing the new triples with P_{new}

```
function CONCORDANCERELATIONS($G_I$, $P_{old}$, $P_{new}$)
    for all $c$ in $\langle P_{old}, \texttt{rdfs:domain}, ?c\rangle(G_{I'})$ do
        $G_{I'} \setminus= \langle P_{old}, \texttt{rdfs:domain}, c\rangle$
        $G_{I'} \cup= \langle P_{new}, \texttt{rdfs:domain}, c\rangle$
    ▷ The following loop only applies to object properties
    for all $c$ in $\langle P_{old}, \texttt{rdfs:range}, ?c\rangle(G_{I'})$ do
        $G_{I'} \setminus= \langle P_{old}, \texttt{rdfs:range}, c\rangle$
        $G_{I'} \cup= \langle P_{new}, \texttt{rdfs:range}, c\rangle$,
    return $G_{I'}$
```

been integrated into the same integration class. Doing so, we guarantee that integrated properties are semantically equivalent as to performing a union operation. Next, we describe how property integration happens and exemplify it by using Example 3. We distinguish two main integration cases for properties: Datatype property and Object property.

Let us start with datatype properties. Following the class-oriented integration idea introduced, we only integrate datatypes if they are part of the same entity, that is, if their domains (e.g., rdfs:Class) have already been integrated into the same :IntegratedResource of type class. Therefore, the invariants for Datatype property integration must reflect this condition. In the following, we present invariant I1 for Datatype property. Note that the remaining invariants should be updated accordingly.

$$
\begin{array}{l}
\forall\langle R_A, R_B, l\rangle \in A_R : \\
\langle R_A, \texttt{rdfs:domain}, C_A\rangle \in G_I \wedge \langle C_A, \texttt{rdfs:subClassOf}, C_{IA}\rangle \in G_I \wedge \\
\langle R_B, \texttt{rdfs:domain}, C_B\rangle \in G_I \wedge \langle C_B, \texttt{rdfs:subClassOf}, C_{IB}\rangle \in G_I \wedge \\
\langle C_{IA}, \texttt{rdf:type}, \texttt{IntegratedResource}\rangle \in G_I \wedge C_{IA} = C_{IB} \wedge \\
\langle R_A, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_I \wedge \\
\langle R_B, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_I \wedge \\
\langle R_A, \texttt{rdf:type}, \texttt{rdf:Property}\rangle \in G_I \wedge \\
\langle R_B, \texttt{rdf:type}, \texttt{rdf:Property}\rangle \in G_I
\end{array}
\implies
\begin{array}{l}
\exists R_l = generateURI(l) : \\
\langle R_l, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \in G_{I'} \wedge \\
\langle R_l, \texttt{rdf:type}, \texttt{rdf:Property}\rangle \in G_{I'} \wedge \\
\langle R_A, \texttt{rdfs:subPropertyOf}, R_l\rangle \in G_{I'} \wedge \\
\langle R_B, \texttt{rdfs:subPropertyOf}, R_l\rangle \in G_{I'}
\end{array}
$$

Algorithm 6 implements the *IntegrateDataTypeProperties* method in Algorithm 4. The implementation of *IntegrateDataTypeProperties* method is similar to Algorithm 2 except for the additional conditions and the manipulation of the domain and range axioms. If the conditions are not fulfilled, the algorithm does not integrate the alignment and preserves it in the set of A_{unused} until their domains are integrated into the same :IntegratedResource in further integrations. Thus, whether this integration will take place depends on the discovered set of alignments from the schema matching approaches. Further, the domain of the :IntegratedResource of type property is an :IntegratedResource of type class. And for the range, we assign the more flexible xsd type (e.g., xsd:string). Algorithm 7 implements the method *concordanceRelations* to ensure all axioms are coherent when replacing an :IntegratedResource or an :IntegratedResource of type property.

For the integration of Object property, we follow a similar approach and only integrate object properties if their domain and range have already been integrated. Accordingly, the object property invariants should reflect this condition and we showcase it for invariant I1 as follows:

Algorithm 8 Join integration

```
Input: $G_I$ is the integrated graph, $P_A$ and $P_B$ are two data type properties, $l$ is a user-provided label for the aligned resource, and $S$ is a user-provided label for the join representation.
Output: Generates the integrated graph $G_{I'}$
function JOININTEGRATION($G_I, P_A, P_B, l, S$)
    $G_{I'} \leftarrow G_I$
    $A_{unused'} \leftarrow A_{unused}$
    $P_l \leftarrow generateURI(l)$
    $P_S \leftarrow generateURI(S)$
    ...                                                        ▷ Invariants I4, I3, I2
    ...                                                        ▷ Start of Invariant I1
    $G_{I'} \cup= \langle P_l, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle$
    $G_{I'} \cup= \langle P_l, \texttt{rdf:type}, \texttt{rdf:Property}\rangle$
    $G_{I'} \cup= \langle P_A, \texttt{rdfs:subPropertyOf}, P_l\rangle$
    $G_{I'} \cup= \langle P_B, \texttt{rdfs:subPropertyOf}, P_l\rangle$
    $C_{rangeA} \leftarrow GetIntegratedClass(G_{I'}, P_A, \texttt{rdfs:range})$
    $G_{I'} \cup= \langle P_l, \texttt{rdfs:range}, C_{rangeA}\rangle$
    $G_{I'} \cup= \langle P_S, \texttt{rdf:type}, \texttt{rdf:Property}\rangle$
    $G_{I'} \cup= \langle P_l, \texttt{:JoinProperty}, P_S\rangle$
    ...                                                        ▷ End of Invariant I1
    return $G_{I'}$
```

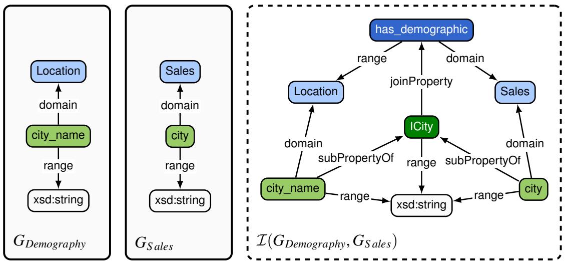

Fig. 12. Example of a Join integration

$$
\begin{array}{lcl}
\forall\langle R_A, R_B, l\rangle \in A_R : & & \\
\langle R_A, \texttt{rdfs:domain}, C_{AD}\rangle \in G_I \wedge \langle C_{AD}, \texttt{rdfs:subClassOf}, C_{IAD}\rangle \in G_I \wedge & & \\
\langle R_B, \texttt{rdfs:domain}, C_{BD}\rangle \in G_I \wedge \langle C_{BD}, \texttt{rdfs:subClassOf}, C_{IBD}\rangle \in G_I \wedge & & \\
\langle C_{IAD}, \texttt{rdf:type}, \texttt{IntegratedResource}\rangle \in G_I \wedge C_{IAD} = C_{IBD} \wedge & & \exists R_l = generateURI(l) : \\
\langle R_A, \texttt{rdfs:range}, C_{AR}\rangle \in G_I \wedge \langle C_{AR}, \texttt{rdfs:subClassOf}, C_{IAR}\rangle \in G_I \wedge & & \langle R_l, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \in G_{I'} \wedge \\
\langle R_B, \texttt{rdfs:range}, C_{BR}\rangle \in G_I \wedge \langle C_{BR}, \texttt{rdfs:subClassOf}, C_{IBR}\rangle \in G_I \wedge & \Longrightarrow & \langle R_l, \texttt{rdf:type}, \texttt{rdf:Property}\rangle \in G_{I'} \wedge \\
\langle C_{IAR}, \texttt{rdf:type}, \texttt{IntegratedResource}\rangle \in G_I \wedge C_{IAR} = C_{IBR} \wedge & & \langle R_A, \texttt{rdfs:subPropertyOf}, R_l\rangle \in G_{I'} \wedge \\
\langle R_A, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_I \wedge & & \langle R_B, \texttt{rdfs:subPropertyOf}, R_l\rangle \in G_{I'} \\
\langle R_B, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle \notin G_I \wedge & & \\
\langle R_A, \texttt{rdf:type}, \texttt{rdf:Property}\rangle \in G_I \wedge & & \\
\langle R_B, \texttt{rdf:type}, \texttt{rdf:Property}\rangle \in G_I & &
\end{array}
$$

The implementation of the Object property integration is very similar to Algorithm 6. We should accordingly add the proper conditions set by the invariants and consider the manipulation of the axioms as we performed for Datatype property. We illustrate the integration of properties in Example 5.

Example 5. *Continuing the Example 4, the algorithm will integrate all* Datatype property *using invariant I1 since their domains were already integrated. If their domains are not integrated, the properties will be preserved in the set of unused alignments A_{unused}. Figure 13 illustrates the integration process of integrating* web_url *and* url*. The integration algorithm uses rdfs:subPropertyOf to connect both* Datatype property *to the* :IntegratedResource *of type property, that is,* URL*. For this example, the following* :IntegratedResource *were generated:* URL*,* Title *and* Name*. Then, we proceed to integrate* Object property *as we can note that the domain and range of* creator *and* participants *have already been integrated, fulfilling the object property requirement. Finally, Figure 14 depicts the complete integrated graph generated from Example 4 and 5. The result of this schema integration generates one* :IntegratedResource *of type class, three* :IntegratedResource *of type datatype property and one* :integratedResource *of type object property. Note that the set A_{unused} contains the alignments not used during this integration, allowing them to be used in further integrations.*

As discussed, our property integration is class-oriented and we only automate the process if the integrated properties are equivalent and can be integrated via a union operator. However, in real practice, this is very restrictive and we allow to integrate two datatype properties whose classes have not been previously integrated in what we call a :JoinProperty. This integration must be performed by a post-process task triggered by a user request. This type of property integration occurs when the domains are not semantically related, but the properties have a semantic correspondence in an input alignment. We thus consider this case as a join operation. Algorithm 8 depicts this integration type. Having no conditions allows us to integrate properties from completely distinct entities. To allow this integration, we must express the join relationship by creating an object property that connects both domain rdfs:Class of

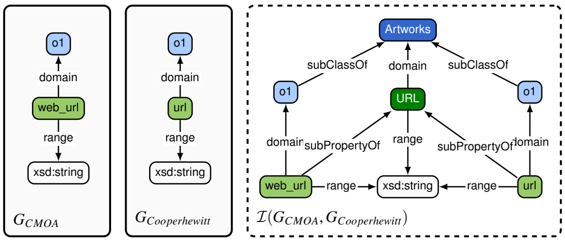

Fig. 13. Result of integrating two classes and two data type properties from G_{CMOA} and $G_{CooperhewittInt}$

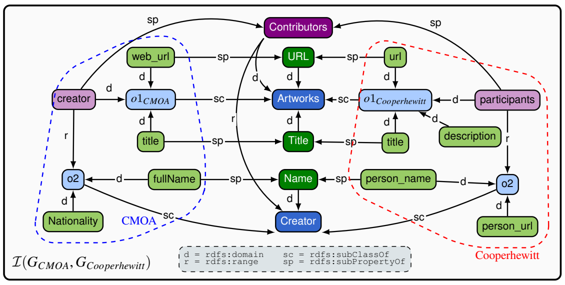

Fig. 14. Final Integration graph from Examples 4 and 5

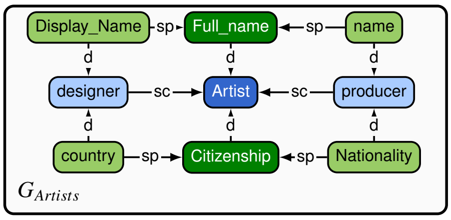

Fig. 15. New data source

$\mathcal{I}(G_{CMOA}, G_{Cooperhewitt})$	$G_{Artists}$	User-provided label l
Creator	Artist	Artist
Nationality	Citizenship	Nationality
Name	Full_name	Name

Table 5

Alignments discovered for $\mathcal{I}(G_{CMOA}, G_{Cooperhewitt})$ and $G_{Artists}$

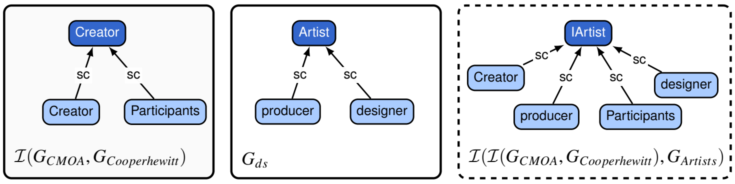

Fig. 16. Result of integrating two integrated classes from $\mathcal{I}(G_{CMOA}, G_{Cooperhewitt})$ and $G_{Artistsi}$

the Datatype properties to integrate, as illustrated in Figure 12. This object property aims to add semantic meaning to the implicit relation of the properties domain. Thus, this Object property connects to the :IntegratedResource using :JoinProperty to identify that this is an on-demand datatype property integration not meeting the regular integration algorithms.

5.2. Incremental example

We will use the final result of Example 5 to perform an incremental integration. For this case, consider the data analyst wants to integrate the typed graph $\mathcal{I}(G_{CMOA}, G_{Cooperhewitt})$ in Figure 14 with the typed graph $G_{artists}$ in Figure 15. In this iteration, the schema matching approach proposed the alignments depicted in Table 5. First, we integrate rdfs:Class. Let us consider the alignment Creator and Artist. Note that we have two :IntegratedResource of type class in this alignment resulting in the use of invariant $I4$ from Algorithm 2. Figure 16 illustrates the integration process for this alignment in which Creator and Artist are replaced by the new :IntegratedResource of type class, namely IArtist . Now, the algorithm proceeds to integrate Datatype property. Let us consider the alignment Nationality and Citizenship representing invariant $I2$. The integration process is illustrated in Figure 17. For this case, instead of creating a new :IntegratedResource of type property, we reuse Citizenship. Therefore, Nationality will be a rdfs:subPropertyOf Citizenship. For the last alignment, Name and Full_Name, we replace both :IntegratedResource of type property by Name. In summary, our approach creates one :IntegratedResource of type class and two :IntegratedResource of type property. The resulting integrated graph is depicted in Figure 18.

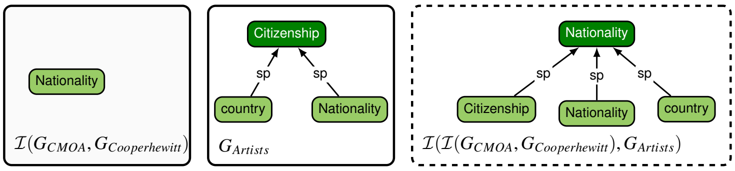
Nationality
$\mathcal{I}(G_{CMOA}, G_{Cooperhewitt})$
Citizenship
sp
sp
country
Nationality
$G_{Artists}$
Nationality
sp
sp
sp
Citizenship
Nationality
country
$\mathcal{I}(\mathcal{I}(G_{CMOA}, G_{Cooperhewitt}), G_{Artists})$

Fig. 17. Result of integrating one integrated data type property from $\mathcal{I}(G_{CMOA}, G_{Cooperhewitt})$ and $G_{Artists}$

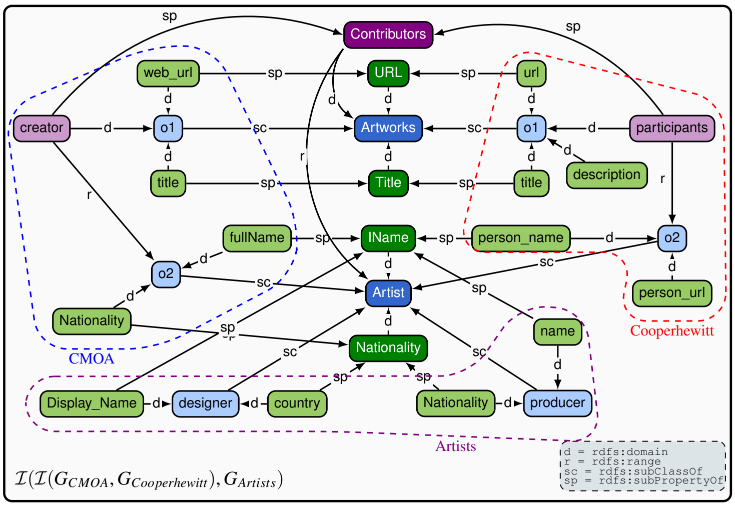

Fig. 18. Integration graph from a second integration

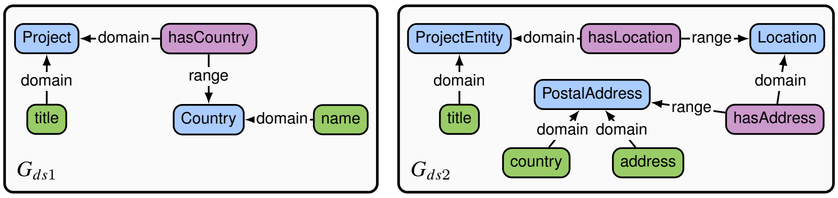

Fig. 19. Example of two different data models

5.3. Limitations

The integration algorithm is our incremental proposal to integrate the typed graphs. However, this algorithm assumes one-to-one input alignments, a limitation all schema alignment techniques have. Our integration algorithm integrates equivalent resources, but it is not able to deal with other complex semantic relationships. To illustrate this, consider the example presented in Figure 19. Let us consider a data analyst interested in retrieving the country's name related to a project. For the G_{ds1} graph, this information can be obtained by using the hasCountry property to retrieve the Country entity. However, the G_{ds2} graph has a different way to represent the country information from a project requiring to retrieve the Location entity, using the hasLocation property, and then using the hasAddress property to retrieve the PostalAddress entity, which contains the country information. The main problem in this case is that the alignment is not one-to-one, instead, the alignment matches a subgraph to one resource. As far as we know, there is no schema alignment technique that considers such case and we for now do not either and leave it for future work. Last, but not least, we would like to stress that even if largely automated, schema integration must be a user-in-the-loop process, since this is the only way to guarantee the input alignments generated by a schema alignment tool are correct.

6. Generation of schema integration constructs

In this section, we show how our approach can be generalized and reused to derive the schema integration constructs of specific virtual data integration systems. We, precisely, instantiate phase ④ in Figure 2. The integrated graph generated in Section 5 contains all relevant metadata about the sources and their integration, which can be used to derive these constructs. Regardless of the virtual data integration system chosen, once the system-specific constructs are created, the user can load them into the system and start wrangling the sources via queries over the global schema. This way, we free the end-user from manually creating such constructs. In the following subsections, we present two methods to generate the schema integration constructs of two representative virtual data integration approaches: mediator-based and ontology-based data access systems

6.1. Mediator-based systems

We consider ODIN as a representative mediator-based data integration system [55]. ODIN relies on knowledge graphs to represent all the necessary constructs for query answering (i.e., the global graph, source graphs and local-as-view mappings). However, all its constructs must be manually created. Thus, we show how to generate its constructs automatically from the integrated graph generated in our approach.

Algorithm 9 shows how to generate the global graph required by ODIN from our integrated graph. In ODIN, the global graph is the integrated view where end-users can pose queries. Hence, this algorithm first merges the subclasses and sub-properties of integrated resources. As a result, we represent the taxonomy of integrated resources with a single integrated resource and properly modify the domain and range of the affected properties. In the case of non-integrated resources, their original definition remains. Figure 20, illustrates the output provided by Algorithm 9 for the generated integration graph in Figure 18. The source schemata and wrappers representing the sources and required by ODIN are immediate to retrieve, since our integration schema preserves the original graph representation bootstrapped per source, and the wrappers remain the same. Finally, since the integration graph was built bottom-up, the local-as-view mappings required by ODIN are generated via the sub-class or sub-property relationships created during schema integration between source schemata elements and integrated elements. All ODIN constructs are therefore straightforwardly generated from the integration graph automatically. Once done, and after loading these constructs into ODIN, the user can query the data sources by querying the global graph of ODIN, which will rewrite the user query over the global graph into a set of queries over the wrappers via its built-in query rewriting algorithm. We have implemented the method here described to generate the constructs and integrated it in ODIN. This implementation is available on this paper's companion website.

6.2. Ontology-based data access systems

Ontology-based data access systems provide a unified view over a set of heterogeneous datasets through an ontology, usually expressed in the OWL 2 QL profile, and global-as-view mappings, often specified using languages such as R2RML and RML. The integrated graph is able to derive an OWL ontology by converting RDFS resources into OWL resources. To this aim, Algorithm 9 requires a post-process step to perform this task. Note that, by definition, the integrated graph contains a small subset of RDFS resources. Thus, we only require to map rdfs:Class to owl:Class and rdf:Property to the corresponding owl:DatatypeProperty or owl:ObjectProperty. Concerning mappings, as long as appropriate algorithms are defined to extract and translate the metadata from the integrated graph, we can derive mappings for any dedicated syntax. Here, we will explain the derivation of RML mappings. Each RML mapping consists of three main components: (i) the Logical Source (rml:logicalSource) to specify the data source, (ii) the Subject Map (rml:subjectMap) to define the class of the RDF instances generated and that will serve as subjects for all RDF triples generated, and (iii) a set of Predicate-Object maps (rml:predicateObjectMap) that define the creation of predicates and its object value for the RDF subjects generated by the Subject Map. Algorithm 10 shows how to generate RML mappings from the integrated graph. The process is as follows.

The algorithm creates RML mappings for each entity defined in a data source schema. Therefore, it iterates over all resources of type rdfs:Class from the integrated graph that are not :IntegratedResource, since those resources were generated from a data source during the bootstrapping phase. For each resource c of type rdfs:Class, we gen-

Algorithm 9 Global schema

Input: An integration graph G_I
Output: A global schema $G_{I'}$

```
function GlobalSchema($G_I$)
    for all $r$ in $\langle ?R, \texttt{rdf:type}, \texttt{:IntegrationResource}\rangle(G_{I'})$ do
        if $\langle r, \texttt{rdf:type}, \texttt{rdf:Property}\rangle(G_{I'})$ then
            for all $s$ in $\langle ?s, \texttt{rdfs:subPropertyOf}, p\rangle(G_{I'})$ do
                $G_{I'} \setminus= \langle s, \texttt{rdfs:domain}, ?domain\rangle(G_{I'})$
                $G_{I'} \setminus= \langle s, \texttt{rdfs:range}, ?range\rangle(G_{I'})$
                $G_{I'} \setminus= \langle s, \texttt{rdf:type}, \texttt{rdf:Property}\rangle$
                $G_{I'} \setminus= \langle s, \texttt{rdfs:subPropertyOf}, p\rangle$
        else if $\langle r, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle(G_{I'})$ then
            for all $s$ in $\langle ?s, \texttt{rdfs:subClassOf}, c\rangle(G_{I'})$ do
                for all $p$ in $\langle ?p, \texttt{rdfs:domain}, s\rangle(G_{I'})$ do
                    $G_{I'} \setminus= \langle p, \texttt{rdfs:domain}, s\rangle$
                    $G_{I'} \cup= \langle p, \texttt{rdfs:domain}, c\rangle$
                for all $p$ in $\langle ?p, \texttt{rdfs:range}, s\rangle(G_{I'})$ do
                    $G_{I'} \setminus= \langle p, \texttt{rdfs:range}, s\rangle$
                    $G_{I'} \cup= \langle p, \texttt{rdfs:range}, c\rangle$
                $G_{I'} \setminus= \langle s, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle$
                $G_{I'} \setminus= \langle s, \texttt{rdfs:subClassOf}, c\rangle$
    return $G_{I'}$
```

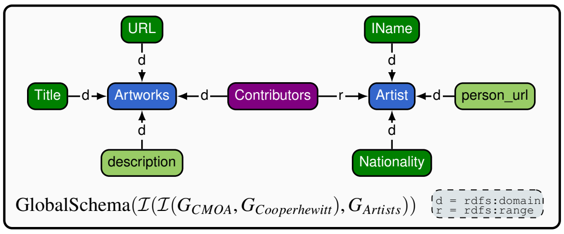

Fig. 20. Generated global schema

erate a unique URI, namely m_c, to represent the RML mapping and create the three RML components. First, we specify the rml:logicalSource for the mapping m_c. Here, we created the required metadata by the Logical Source: (i) the rml:source to specify the data source location, which is obtained from the source wrapper through the method $GetDataSourcePath(c)$, (ii) the rml:referenceFormulation to express the data source format, which is assigned dynamically depending on the data source type by using the method $GetDataSourceType(c)$ (e.g., for CSV and JSON it generates ql:CSV or ql:JSONPath), and (iii) the rml:iterator to define the iteration pattern to retrieve each data instance to be mapped, obtained from the wrapper definition using the method $GetIterator(c)$.

The next step is to specify the rml:subjectMap for the mapping m_c. Here, we define the variable $c_{targetType}$ using the URI from c to define the subject type of all RDF instances produced. If resource c is a subclass of an :IntegratedResource, we define $c_{targetType}$ as the integrated URI. Then, we created the required metadata for the rml:subjectMap: (i) the rr:template to define the URIs of the instances, for which we use the URI from $c_{targetType}$ and append a unique identifier from the data source, and (ii) the rr:class to specify the class of the subjects produced. Then, we create of the set of rml:predicateObjectMap. To that end, we iterate over all rdf:Property that has as domain the class c. Then, for each property p, we define the variable $p_{predicate}$ as the URI from p. If p is a subproperty of an :IntegratedResource or p is a subproperty of an :IntegratedResource that represents a :JoinProperty, we define $p_{predicate}$ as the integrated URI or as the JoinProperty URI, respectively. Then, we generate the rml:predicateObjectMap metadata. Here, we distinguish two cases: properties that are not part of a :JoinProperty and those that are. In the first case, we will generate the metadata as follows: (i) the rr:predicate to indicate that property $p_{predicate}$ is used to map the subject with the object, and (jj) the rr:objectMap as well as rml:reference to indicate which element of the data source schema should be used to generate the RDF objects instances. Here, we use the method $GetSourceReference(p)$ to retrieve the reference of the data source (e.g, a column or JSON key) from a given property, which is obtained from the wrapper metadata. Finally, if the property is part of a :JoinProperty, we will create a rml:predicateObjectMap for all properties that are part of the :JoinProperty. Then, the following metadata is generated for each rml:predicateObjectMap: (i) the rr:predicate where we use the join property URI to connect two entities from different sources, and (ii) the rr:objectMap along with a rml:parentTriplesMap, which is used to link the mappings of two different entities. We use the method $GetClassMapping(p')$ to get the mapping URI of a given property. Finally, we proceed to create the metadata for the rml:parentTriplesMap: (i) the rr:joinCondition containing the rr:child and the rr:parent, which require a reference of an element from the data source to create the join. We used the method $GetSourceReference$ to obtain the references of each property p and p'.

As a result, the provided algorithm generates the RML mappings for all entities and data sources contained in an integrated graph and uses the annotations with regard to unions and joins to construct global-as-view mappings according to the global schema (e.g., ontology). Last but not least, this algorithm showcases the feasibility of gen-

Algorithm 10 RML mappings generation

```
Input: An integration graph $G_I$
Output: RML mappings
function GENERATE RML_MAPPINGS($G_I$)
    $G_M \leftarrow \emptyset$
    for all $c$ in $\langle ?C, \texttt{rdf:type}, \texttt{rdfs:Class} \rangle \in (G_I) \wedge \langle ?C, \texttt{rdf:type}, \texttt{:IntegratedResource} \rangle \notin (G_I)$ do
        $m_c \leftarrow generateURI(c)$
        $G_M \cup= \langle m_c, \texttt{rdf:type}, \texttt{rr:TriplesMap} \rangle$
        $b_1 \leftarrow \textsc{BlankNode}()$
        $G_M \cup= \langle m_c, \texttt{rml:logicalSource}, b_1 \rangle$
        $G_M \cup= \langle b_1, \texttt{rml:source}, GetDataSourcePath(c) \rangle$
        $G_M \cup= \langle b_1, \texttt{rml:referenceFormulation}, GetDataSourceType(d) \rangle$
        $G_M \cup= \langle b_1, \texttt{rml:iterator}, GetIterator(d) \rangle$
        $c_{targetType} \leftarrow c$
        if $\langle c, \texttt{rdfs:subClassOf}, R_I \rangle \in (G_I) \wedge \langle R_I, \texttt{rdf:type}, \texttt{:IntegratedResource} \rangle \in (G_I)$ then
            $c_{targetType} \leftarrow R_I$
        $b_2 \leftarrow \textsc{BlankNode}()$
        $G_M \cup= \langle m_c, \texttt{rr:subjectMap}, b_2 \rangle$
        $c_{ID} \leftarrow GenerateID(c)$
        $G_M \cup= \langle b_2, \texttt{rr:template}, c_{targetType} + "/" + c_{ID} \rangle$
        $G_M \cup= \langle b_2, \texttt{rr:class}, c_{targetType} \rangle$
        for all $p$ in $\langle ?P, \texttt{rdf:type}, \texttt{rdf:Property} \rangle \in (G_I) \wedge \langle ?P, \texttt{a}, \texttt{:IntegratedResource} \notin (G_I) \wedge \langle ?P, \texttt{rdfs:domain}, \texttt{c} \rangle \in (G_I)$ do
            $p_{predicate} \leftarrow p$
            if $\langle p, \texttt{rdfs:subPropertyOf}, R_I \rangle \in (G_I) \wedge \langle R_I, \texttt{a}, \texttt{:IntegratedResource} \rangle \in (G_I) \wedge \langle R_I, \texttt{:JoinProperty}, R_{join} \rangle \notin (G_I)$ then
                $p_{predicate} \leftarrow R_I$
            else if $\langle p, \texttt{rdfs:subPropertyOf}, R_I \rangle \in (G_I) \wedge \langle R_I, \texttt{a}, \texttt{:IntegratedResource} \rangle \in (G_I) \wedge \langle R_I, \texttt{:JoinProperty}, R_{join} \rangle \in (G_I)$ then
                $p_{predicate} \leftarrow R_{join}$
            if $\langle p, \texttt{:JoinProperty}, p_{predicate} \rangle \notin (G_I)$ then
                $b_3 \leftarrow \textsc{BlankNode}()$
                $G_M \cup= \langle m_c, \texttt{rr:predicateObjectMap}, b_3 \rangle$
                $G_M \cup= \langle b_3, \texttt{rr:predicate}, p_{predicate} \rangle$
                $b_4 \leftarrow \textsc{BlankNode}()$
                $G_M \cup= \langle b_3, \texttt{rr:objectMap}, b_4 \rangle$
                $G_M \cup= \langle b_4, \texttt{rml:reference}, GetSourceReference(p) \rangle$
            else
                for all $\langle p', \texttt{rdfs:subPropertyOf}, R_I \rangle \in (G_I) \wedge \langle R_I, \texttt{:JoinProperty}, p_{predicate} \rangle \notin (G_I) \wedge p \neq p'$ do
                    $b_3 \leftarrow \textsc{BlankNode}()$
                    $G_M \cup= \langle m_c, \texttt{rr:predicateObjectMap}, b_3 \rangle$
                    $G_M \cup= \langle b_3, \texttt{rr:predicate}, p_{predicate} \rangle$
                    $b_4 \leftarrow \textsc{BlankNode}()$
                    $G_M \cup= \langle b_3, \texttt{rr:objectMap}, b_4 \rangle$
                    $G_M \cup= \langle b_4, \texttt{rr:parentTriplesMap}, GetClassMapping(p') \rangle$
                    $b_5 \leftarrow \textsc{BlankNode}()$
                    $G_M \cup= \langle b_4, \texttt{rr:joinCondition}, b5 \rangle$
                    $G_M \cup= \langle b_5, \texttt{rr:child}, GetSourceReference(p) \rangle$
                    $G_M \cup= \langle b_5, \texttt{rr:parent}, GetSourceReference(p') \rangle$
    return $G_{I'}$
```

erating RML mappings from the integrated graph. Note that the generated RML mappings can be used for any RML compliant engines (virtual or materialized) such as Morph-RDB [56], RDFizer [57] and RMLStreamer [58]. Importantly, note that our objective is not to generate optimized RML mappings, which should be part of the future work.

7. Evaluation

In this section, we evaluate the implementation of our approach. To that end, we have developed a Java library named Nextia$_{\text{DI}}$, which incorporates the algorithms described in previous sections. In order to scrutinize the added value of our approach and guide the evaluation activities, we have compiled the following set of research questions:

- (Q1) Does the usage of Nextia$_{\text{DI}}$ reduce the effort devoted to schema integration?
- (Q2) Does the usage of Nextia$_{\text{DI}}$ reduce the time devoted to schema integration?
- (Q3) Does the quality of the generated integrated schema improve using Nextia$_{\text{DI}}$?
- (Q4) Does the runtime of Nextia$_{\text{DI}}$ scale with an increasing volume of data?

To address questions Q1, Q2, and Q3, we conducted a user study. Note we distinguish between Q1 and Q2, since the first is meant to report on the qualitative perception (according to the feedback received), while Q2 is a quantitative metric independent of the participant feelings during the activity. Regarding Q4, we carried out scalability experiments. Nextia$_{\mathrm{DI}}$, as well as all details and reproducibility instructions are available on the companion website[12].

7.1. User study

This user study aims at evaluating the efficiency and quality of Nextia$_{\mathrm{DI}}$ in automatically supporting the task of schema integration compared to a conventional schema integration pipeline. The study is, hence, divided into three tasks: (i) generation of source schemata, (ii) generation of an integrated schema, and (iii) generation of mappings. Participants were asked to perform each task using both pipelines: the conventional approach and supported by Nextia$_{\mathrm{DI}}$. In the following, we describe our process:

Data sources We selected four data sources collected from the Tate collection[13] and CMOA collection[14]. All data sources are modeled using JSON and contain distinct entities related to artworks and artists. The number of schema elements in each data source, are, respectively, 48, 30, 30, and 13.

Definition of the data collection methods. As discussed in Section 2, no tool from the state-of-the-art covers the same end-to-end process as our approach does. Thus, as a baseline representation of the conventional schema integration pipeline, we designed a set of notebooks to support each of the tasks described above[15]. Such notebooks contain detailed instructions on how to use the RDFLib Python library for the generation of RDF data. We leveraged on our industrial partners to generate a realistic baseline. It is important to observe that CoMerger[39] could be used as part of the pipeline for schema integration, yet the lack of proper documentation and the large number of issues at execution time hindered its use. In order to evaluate Nextia$_{\mathrm{DI}}$, we integrated its functionalities into the previously described virtual data integration tool ODIN (see Section 6). Thus, providing an intuitive user interface to use Nextia$_{\mathrm{DI}}$ functionalities: bootstrapping, incremental schema integration and derivation of constructs. To prevent any bias, the study is conducted in two formats: *v1)* participants used first Nextia$_{\mathrm{DI}}$ followed by the conventional approach; and *v2)* first performing the conventional approach and followed by Nextia$_{\mathrm{DI}}$. At the end of each task, a post-questionnaire is provided. Overall, participants were given at most 2 hours to implement each of the pipelines, although as described later, this was not sufficient for some participants when implementing the conventional pipeline.

Selection of participants. A set of 16 practitioners was selected to participate in the user study. Care was taken in selecting participants both from academia (researchers at UPC) and industry with different backgrounds and expertise (e.g., a broad range of skills, different seniority levels). All of them have participated in at least one Data Science project. Figure 21 summarizes how participants describe themselves with regard to the relevant skills required to perform the study. Specifically, we asked them to rate how skilled they felt in data integration (and specifically in the sub-domains of schema integration and virtual data integration systems), data modeling and knowledge graphs. All these skills were needed during the study and, as shown later, provide value when interpreting the results obtained. Participants were divided into two equally sized groups and assigned either *v1* or *v2*.

Evaluation of collected results. In order to evaluate the results of each survey, we compare the aggregated answers for each non free-form question distinguishing between the conventional approach and Nextia$_{\mathrm{DI}}$. Additionally, in order to compare the answers to questions with different scales, we have normalized all aggregated values to the range $[0, 1]$ using the MinMax normalization technique. For all surveys, we measure the time spent on the task, and whether participants managed to finish it or not. We also surveyed participants with questions related to their perceived effort in completing each task. Furthermore, for each task, we also compute the *soundness* and *complete-*

[12] https://www.essi.upc.edu/dtim/nextiadi/

[13] https://github.com/tategallery/collection

[14] https://github.com/cmoa/collection

[15] The notebooks and all experiments data are available on the companion website of this paper

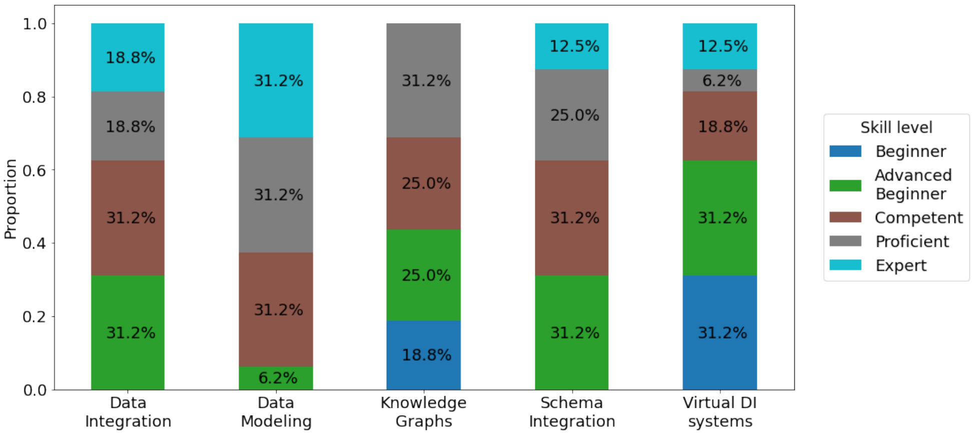

Fig. 21. Distribution of level across skills for all participants

ness of the produced results. Soundness is computed by manually checking the correctness of the generated results (e.g., typing and domain and ranges are correctly defined), while completeness is computed using the expression $C = 1 - i_i/i_t$, where i_i is the number of incomplete items and i_t the number of total items, as defined in [59]. This way, we avoid any kind of bias by selecting external metrics defined in the literature.

7.2. Analysis of results

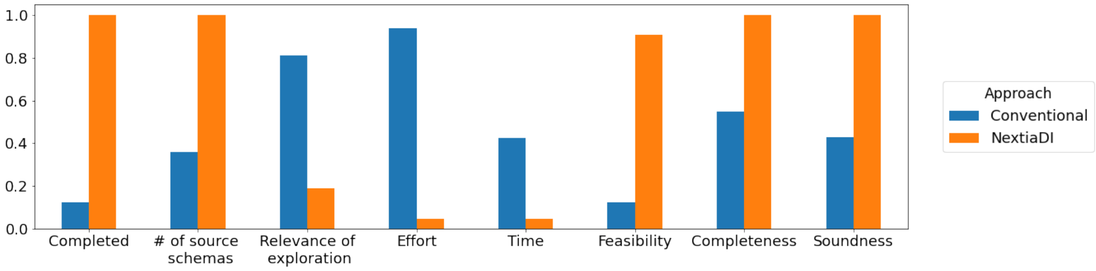

Fig. 22. Aggregated results for the bootstrapping phase. *Completed* specifies whether participants finished the task within the available time (higher is better). *# of source schemas* denotes how many schema elements participants managed to bootstrap in the available time (higher is better). *Relevance of exploration* denotes the need a participant perceived to explore the structure of the sources to conduct the activity (lower is better). *Effort* and *Time* specifies, respectively, the perceived effort and time to complete the task (lower is better). *Feasibility* is used to quantify the perceived feasibility of running the task with the approach at hand (higher is better). *Completeness* and *Soundness* refer to the quality metrics previously introduced.

Results on bootstrapping. Figure 22 shows the aggregated results from the bootstrapping survey, where participants were required to define a graph-based schema for each data source with the two approaches. In the conventional approach, only 12.5% of the participants generated the four schemata, while using Nextia$_{\mathrm{DI}}$ all participants completed the task. We observe that the main challenge in the conventional approach is the required domain knowledge in data modeling and knowledge graphs. Participants P2 and P9 said: *"It is really hard and requires knowledge of specific syntax and technologies such as Semantic Web, Knowledge Graph, Data Modeling."* and *"You need expertise in Knowledge Graphs and Modeling (e.g., in RDF) to come up with sound schemas/models."* Regardless of their background, all participants in the conventional approach reported a high level of effort. The main reason is the need to explore the data sources, as reported by 81.3% of the participants who reported they were constantly exploring the content of the data sources to understand the structure to design the schema. With Nextia$_{\mathrm{DI}}$, the efforts reported were smoother and 87.5% said no effort is needed. Regarding the time, participants spent an average of

53.44 mins bootstrapping the sources in the conventional approach, while using Nextia$_{\mathrm{DI}}$, participants completed the task in an average of 5.9 mins. As participant P5 pointed out, *"It provided a great automation of the process, reducing the time spent to generate the source schema."* When asked how feasible is to bootstrap the schema of data sources using the conventional approach, all participants reported it as unfeasible. With Nextia$_{\mathrm{DI}}$, 93.8% reported it as feasible. The only person not assessing Nextia$_{\mathrm{DI}}$positively commented *"Actually I would like to know how Nextia$_{DI}$behaves when importing large schemas."* However, this is a laborious task we could not cover in the study, which we had to keep reasonable (below 2h of effort) to involve as many practitioners as possible. We report on this aspect when discussing Q4 later. In this sense, as P3 and P4 reported: *"In the context of big data, automation is much required. Nextia$_{DI}$ provides this."* and *"Having an automatic way of generating those schemas without even having to look at the underlying data, is a very good solution when huge amounts of data coming from different structures are present."* Concerning the quality of the bootstrapped source schemata, overall by means of the conventional approach they yielded 23 schemas, yet only 21.74% completely represent the data source schema and 47.82% of the schemas only represent less than 31% of the data source content. Additionally, with regard to soundness, only 43.48% of the schemas were compliant with the metamodel. Most common mistakes were the lack of typing (e.g., class or properties) and incorrect usage of axioms. The schemata with the best soundness and completeness were produced by participants who are proficient in knowledge graphs and competent to experts in data modeling. In the case of Nextia$_{\mathrm{DI}}$, all their schemata are complete with regard to the data source content and are compliant with the metamodel.

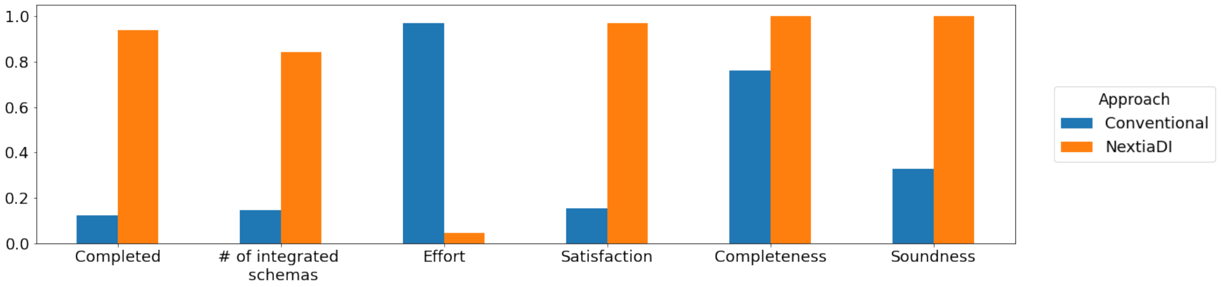

Fig. 23. Aggregated results for the schema integration phase. *Completed* specifies whether participants finished the task within the available time (higher is better). *# of integrated schemas* denotes how many schema elements participants managed to integrate in the available time (higher is better). *Effort* specifies perceived effort to complete the task (lower is better). *Satisfaction* is used to quantify the perceived satisfaction of running the task with each approach (higher is better). *Completeness* and *Soundness* refer to the quality metrics previously introduced.

Results on schema integration. Figure 23 depicts the aggregated results for the schema integration survey, where participants were required to integrate the schemas generated in the previous task. In order to accomplish it, three main integrated schemata had to be created (i.e., DS1-DS2, DS1-DS2-DS3, and DS1-DS2-DS3-DS4). On the conventional approach, only 6.25% of the participants managed to generate the three integrated schemas. In contrast, using Nextia$_{\mathrm{DI}}$ the task completion rates are significantly higher, although one participant did not finish the task either. In terms of effort, 87.5% of the participants stress that this task needed considerable effort. As highlighted by participants P5 and P11 *A lot of different classes, properties, sequences... from different schemas have to be taken into account, and it is difficult to see the big picture when there is a lot of data and integrate without making mistakes.*; and *it is hard to ensure everything makes sense to merge elements*, respectively. Compared to Nextia$_{\mathrm{DI}}$, 81.3% of the participants said no effort was required. Concerning the time, participants spent an average of 30.06 minutes in the conventional approach. Some participants reported frustration with this task: P1 spent 45 minutes generating one integrated schema, while P15 spent 50 minutes and yet could not complete it. In contrast, participants spent an average of 6.94 minutes with Nextia$_{\mathrm{DI}}$. As participant P1 discusses *As the number of class or properties increases, it becomes harder and harder to manually check every possible pair of properties.* When asked if the conventional approach for schema integration is feasible, 31.3% of participants reported not feasible. In this line, participants P8 and P10 argument that *On top of having to develop the base schemas, having to manually integrate all the elements that are needed to is impossible in a big data context, as the amount of time needed is very large. Additionally, the* *chances of making a mistake are considerable* and *The complexity of real datasets makes the integration process not trivial and not accurate.* Alternatively, using Nextia$_{\text{DI}}$, 75% of participants reported that schema integration is feasible. As P1 commented *since the process is done automatically from the beginning, there is a coherence between how the sources are constructed, making it easy to maintain the elements pointing from the source to the global schema.* Concerning quality only 22.22% of the integrated schemata using the conventional approach represent all integrated concepts. Regarding soundness, only 33.33% are compliant with the metamodel. When asked about the participants' satisfaction, 87.5% are very satisfied and 12.5% satisfied with the results provided by Nextia$_{\text{DI}}$. Some of the comments included *it seems to use best practices in modeling, for modeling the global schema*; *Effortless process. Semantically sound schemas*; and *it is so easy and it completely merge the entities while keeping the connection to the original schemas.* Relevantly, one data modeling expert realized Nextia$_{\text{DI}}$ is not able to integrate elements at different granularity (as discussed in Section 5.3). Yet, the participant acknowledged the value of Nextia$_{\text{DI}}$'s output as a baseline to build a better solution.

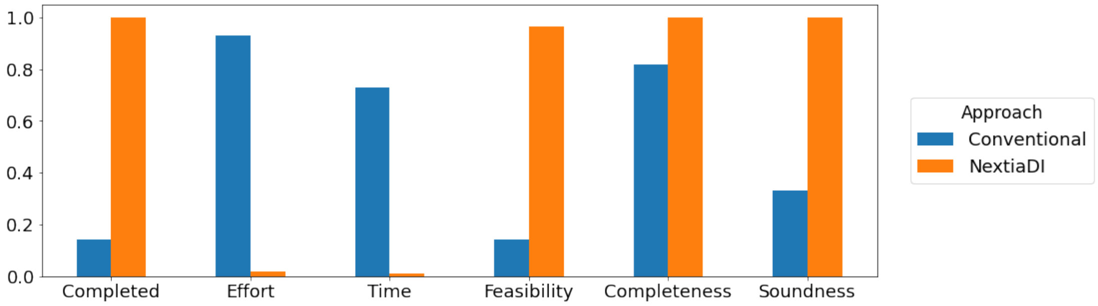

Fig. 24. Aggregated results for the mapping generation phase. *Completed* specifies whether participants finished the task within the available time (higher is better). *Effort* and *Time* indicate, respectively, the perceived effort and time to complete the task (lower is better). *Feasible* is used to quantify the perceived feasibility of the task in Big Data settings (higher is better). *Completeness* and *Soundness* refer to the quality metrics previously introduced.

Results on generation of mappings. Figure 24 shows the aggregated results for the mapping generation survey, where participants were asked to generate the mappings between the source schemata generated in step 1 and the integrated schema generated in step 2. In the conventional approach, only 37.5% of the participants managed to generate the mappings. Indeed, most participants reported a high level of effort required, while all participants generated the mappings using Nextia$_{\text{DI}}$ with 93.8% of them reporting no effort was required. Concerning time, the conventional approach required an average of 20.81 mins for this task. As participant P2 discusses *"mapppings as same as schema integration requires a lot of time to verify everything is correct."* In contrast, using Nextia$_{\text{DI}}$ the process is immediate due to its bottom-up nature (see Section 6). When asked how feasible is to generate mappings using the conventional approach all participants reported its unfeasibility. Oppositely, using Nextia$_{\text{DI}}$ 81.3% and 18.8% said it is very feasible or feasible. Participants emphasize that automation is crucial, precisely P5 commented *Nextia$_{DI}$ has a great potential to make Data Integration feasible in Big Data scenarios, as it generates the mappings automatically.* and *"The system creates the mappings automatically and keeps track of them. In addition it allows to modify/remove the incorrect ones. So, it is quite feasible in a Big Data context"*, respectively. Regarding quality, with the conventional approach, 6 mappings files were generated among all participants, being only 33.33% fully complete and sound.

Conclusion. We confirmed that Nextia$_{\text{DI}}$ reduced the effort perceived to bootstrap, integrate the data sources and generate mappings (Q1), which is also confirmed by the time reported in all three tasks (Q2). The study also shows that the effort/time required correlates with the size of the schemata to be produced and therefore the number of elements to check. The larger the number of comparisons, the higher the frustration and perceived effort by the participants, which confirms the need for automating such processes. About the quality of the results produced (Q3), Nextia$_{\text{DI}}$ automates bootstrapping based on rules at the metamodel level, which guarantees the correctness of the bootstrapped schema from a data modeling point of view. Similarly, the schema integration is based on solid data modeling principles (e.g., based on taxonomies of concepts) that guarantee good quality outputs. This is confirmed by all participants (specially relevant to the positive answer from all data modeling experts in both tasks). Finally, generating mappings is completely transparent in Nextia$_{\mathrm{DI}}$, which is highly appreciated, while the conventional approach requires expertise on data integration to properly conduct the task. All participants were satisfied with the quality of the outputs of Nextia$_{\mathrm{DI}}$ and uncertain about the quality of the output they generated in the conventional approach. Importantly, their answers are sound with the qualitative metrics (completeness and soundness) computed and shown in the three figures.

Another relevant conclusion of the experiment, raised by most of the participants, is the difficulty of the tasks at hand due to the required combined skills. In the conventional approach, they spent a considerable amount of time exploring the sources, modeling and expressing schemata and mappings in RDFS. This combined profile is difficult to find in the field of Data Science, specially, knowledge graph experts as shown in Figure 21. This fact confirms our claim that is not feasible to make the users responsible for generating the schema integration constructs of virtual data integration systems, even for small scenarios. This can explain the low impact of such tools in industry. Fortunately, all participants believe Nextia$_{\mathrm{DI}}$ is an improvement to existing approaches.

7.3. Scalability experiments

In order to address research question Q4, we evaluate our two technical contributions (i.e., bootstrapping and schema integration) to assess their computational complexity and runtime performance. All experiments were carried out on a Mac Intel core 2.3 GHz i5 processor with 16GB RAM and Java compiler 11.

7.3.1. Evaluation of Bootstrapping

We evaluated the bootstrapping of JSON and CSV data sources by measuring the impact of the schema size. Therefore, we increase the size of the schema elements. This experiment was executed 10 times. We describe the dataset preparation and the results obtained in the following.

Dataset preparation. We generated 100 datasets in JSON and CSV format. The initial JSON dataset contains a schema of 100 keys (9 objects and 91 attributes). We incrementally increased the number of schema elements by appending 50 new keys (1 object with 49 attributes) in the schema. For example, the second and third datasets will contain 150 keys (10 objects and 140 attributes) and 200 keys (11 objects and 189 attributes), respectively. For the initial CSV dataset, the schema contains 91 headers, and we incrementally appended 49 new headers.

Results. Figure 25 depicts the correlation between time in milliseconds and the number of keys/headers. Note that in both data sources, the time to generate a typed graph schema depends on the size of the schema. JSON bootstrapping requires more time than CSV bootstrapping, since the algorithm will parse one instance of the JSON to extract the JSON schema and apply the production rules. The initial dataset took 27 milliseconds to produce a typed graph, while the last dataset, with a schema of 108 objects and 4942 attributes, took 71 milliseconds. In contrast, CSV bootstrapping only requires the header information to produce a typed graph schema. The initial dataset took 3 milliseconds to produce the typed graph, and the last dataset, with a schema of 4942 headers, took 17 milliseconds. Overall, we can observe some peeks in the trend. However, we consider these peaks are anomalies produced due to the java garbage collector performance, since all results are constant within milliseconds. The performance obtained by the JSON and CSV shows we can rapidly bootstrap the schema in data wrangling tasks.

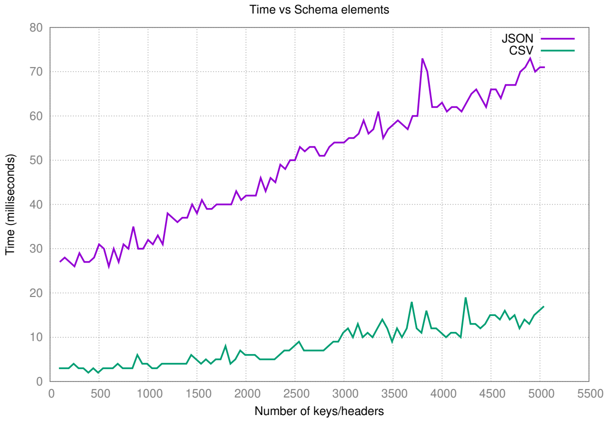

Fig. 25. Performance evaluation when the number of schema elements increased

7.3.2. *Evaluation of schema integration*

We evaluate the schema integration under three scenarios: i) increasing the number of alignments in an incremental integration and ii) integrate a constant number of alignments with a growing number of elements in the schemata and iii) perform integration using real data.

Experiment 1 — increased alignments Here, the algorithm requires the generation of schemas and alignments that will be integrated incrementally. For the schemata, we reuse the typed graph generated by the initial JSON dataset generated in Section 7.3.1 to generate 100 typed graph schemata. All the resulting schema contain 9 classes, 90 data type properties and 1 object property. We generated the alignments between each pair of typed graph by increasing in one the number of alignments with regard to the previous pair. Therefore, the first integration generates 1 alignment and the last iteration 100 alignments. Figure 26 depicts the correlation between time in milliseconds and the number of alignments. We can observe that the time to integrate schemata depends on the number of alignments. The time to integrate one alignment took two milliseconds, while the integration of 100 alignments took 112 milliseconds. Moreover, in iteration 13, the number of integrated classes converged. Then, the remaining iteration reuses all integrated classes by applying invariant I3 from Algorithm 2. For datatype and object properties, they converged in iterations 35 and 4, respectively. Then, the remaining iterations did not create any new integrated properties. In addition, the final integrated graph contains 1000 classes, 9 integrated classes, 9000 datatype properties, 90 integrated datatype properties, 900 object properties, and 1 integrated object property. Overall, the algorithm efficiently integrates schemas incrementally.

Experiment 2 — increased number of schema elements. Here, we generated 100 schemas using JSON datasets where the number of elements in the schema were increased by 50 keys in each iteration. For the generation of alignments, we produced 100 alignments in each iteration since the goal is to measure the impact of the schema size in the integration. Figure 27 depicts the correlation between time in milliseconds and the total number of resources (e.g., classes and properties) in the integrated graph. We observe that the size of the integrated graph impact the time for integrating alignments. The final integrated graph contains 5950 classes, 9 integrated classes, 256500 datatype properties, 90 integrated datatype properties, 5850 object properties and 1 integrated object property. In total 268400 resources. The impact on the time due to the graph size is largely due to how Jena manages the graph in memory (the underlying triplestore used in the experiments). This demonstrates that Nextia$_{\mathrm{DI}}$ can handle the integration and propagation of changes faster, even when the integrated schema is large.

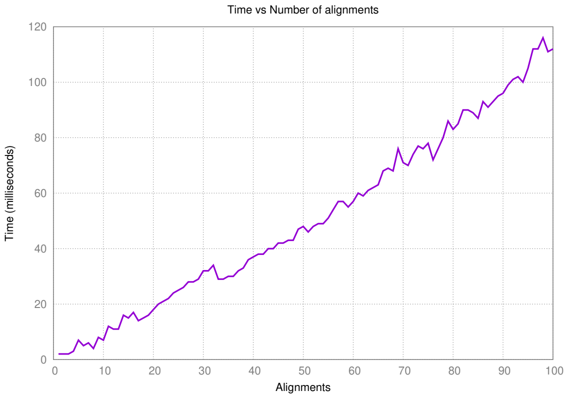

Fig. 26. Performance evaluation when alignments increased incrementally

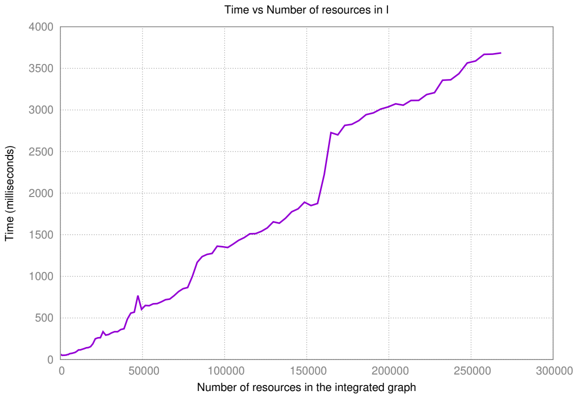

Fig. 27. Performance evaluation when the number of schema elements increased

Experiment 3 — real data. We have selected two tracks of the Ontology Alignment Evaluation Initiative: anatomy and large biomed track. The former contains the Foundational Model of Anatomy (FMA) with 2744 classes and is part of the National Cancer Institute Thesaurus (NCI) with 3304 classes. The alignments provided are 1516 class alignments. The latter contains the FMA with 78998 classes and 15 properties, and NCI with 77269 classes and 186 properties. There are 2686 class alignments for this track. This integration was performed in one step. For the anatomy track, all elements were integrated in 120 milliseconds. The final integrated graph contains 6048 classes and 1516 integrated classes. For the large biomed track, all elements were integrated in 3043 milliseconds with 156267 classes, 2686 integrated classes, 78 data type properties and 123 object properties. Overall, our integration approach is efficient when dealing with large schemas, such as the biomed track.

8. Conclusions and future work

This paper presents an approach for efficiently bootstrapping schemata of heterogeneous data sources and incrementally integrating them to facilitate the generation of schema integration constructs in virtual data integration settings. This process is specially thought to meet the requirements of data wrangling as required in Big Data scenarios: i.e., highly heterogeneous data sources and dynamic environments. As such, our proposal deal with heterogeneous data sources, follows an incremental approach to follow a pay-as-you-go integration approach and largely automates the process. Relevantly, our approach is not specific for a given system and it generates system-agnostic metadata that can be later used to generate the specific constructs of the most relevant virtual data integration systems. Last but not least, we have presented Nextia$_{\mathrm{DI}}$, an open source tool implementing our approach. We have also shown the added value of our proposal by conducting a user study with 16 practitioners and scalability experiments. Both the user study and experiments show the value of our approach, which covers a gap not covered by any other approach before.

This work opens many interesting research lines from it. For example, how to generalize the current approach to integrate complex semantic relationships between schemas (beyond one-to-one mappings) or develop a hybrid approach that, once the integrated schema is generated in a bottom-up approach, it allows the user to enrich the automatically generated outputs in a top-down approach. The ultimate question we would like to address is how to use these techniques to suggest refactoring techniques over the underlying data sources to facilitate their alignment and integration.

Acknowledgements. This work was partly supported by the DOGO4ML project, funded by the Spanish Ministerio de Ciencia e Innovación under project PID2020-117191RB-I00, and D3M project, funded by the Spanish Agencia Estatal de Investigación (AEI) under project PDC2021-121195-I00. Javier Flores is supported by contract 2020-DI-

027 of the Industrial Doctorate Program of the Government of Catalonia and Consejo Nacional de Ciencia y Tecnología (CONACYT, Mexico). Sergi Nadal is partly supported by the Spanish Ministerio de Ciencia e Innovación, as well as the European Union - NextGenerationEU, under project FJC2020-045809-I.

References

[1] W.A. Günther, M.H.R. Mehrizi, M. Huysman and F. Feldberg, Debating big data: A literature review on realizing value from big data, *J. Strateg. Inf. Syst.* **26**(3) (2017), 191–209.

[2] S. Kandel, J. Heer, C. Plaisant, J. Kennedy, F. van Ham, N.H. Riche, C.E. Weaver, B. Lee, D. Brodbeck and P. Buono, Research directions in data wrangling: Visualizations and transformations for usable and credible data, *Inf. Vis.* **10**(4) (2011), 271–288.

[3] P. Jovanovic, S. Nadal, O. Romero, A. Abelló and B. Bilalli, Quarry: A User-centered Big Data Integration Platform, *Inf. Syst. Frontiers* **23**(1) (2021), 9–33. doi:10.1007/s10796-020-10001-y.

[4] A.Y. Halevy, A. Rajaraman and J.J. Ordille, Data Integration: The Teenage Years, in: *VLDB*, ACM, 2006, pp. 9–16.

[5] S. Kandel, A. Paepcke, J.M. Hellerstein and J. Heer, Enterprise Data Analysis and Visualization: An Interview Study, *IEEE Trans. Vis. Comput. Graph.* **18**(12) (2012), 2917–2926.

[6] P. Pereira, J. Cunha and J.P. Fernandes, On Understanding Data Scientists, in: *VL/HCC*, IEEE, 2020, pp. 1–5.

[7] B. Golshan, A.Y. Halevy, G.A. Mihaila and W. Tan, Data Integration: After the Teenage Years, in: *PODS*, ACM, 2017, pp. 101–106.

[8] D. Abadi, A. Ailamaki, D.G. Andersen, P. Bailis, M. Balazinska, P.A. Bernstein, P.A. Boncz, S. Chaudhuri, A. Cheung, A. Doan, L. Dong, M.J. Franklin, J. Freire, A.Y. Halevy, J.M. Hellerstein, S. Idreos, D. Kossmann, T. Kraska, S. Krishnamurthy, V. Markl, S. Melnik, T. Milo, C. Mohan, T. Neumann, B.C. Ooi, F. Ozcan, J.M. Patel, A. Pavlo, R.A. Popa, R. Ramakrishnan, C. Ré, M. Stonebraker and D. Suciu, The Seattle report on database research, *Commun. ACM* **65**(8) (2022), 72–79.

[9] M. Stonebraker and I.F. Ilyas, Data Integration: The Current Status and the Way Forward, *IEEE Data Eng. Bull.* **41**(2) (2018), 3–9.

[10] X.L. Dong and D. Srivastava, *Big Data Integration*, Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2015.

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

[12] M. Lenzerini, Data Integration: A Theoretical Perspective, in: *PODS*, ACM, 2002, pp. 233–246.

[13] A. Halevy, M. Franklin and D. Maier, Principles of dataspace systems, in: *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, ACM, 2006, pp. 1–9.

[14] L.M. Haas, Beauty and the Beast: The Theory and Practice of Information Integration, in: *ICDT*, Lecture Notes in Computer Science, Vol. 4353, Springer, 2007, pp. 28–43.

[15] N.W. Paton, K. Belhajjame, S.M. Embury, A.A.A. Fernandes and R. Maskat, Pay-as-you-go Data Integration: Experiences and Recurring Themes, in: *SOFSEM*, Lecture Notes in Computer Science, Vol. 9587, Springer, 2016, pp. 81–92.

[16] P. Atzeni, P. Cappellari, R. Torlone, P.A. Bernstein and G. Gianforme, Model-independent schema translation, *VLDB J.* **17**(6) (2008).

[17] J. Flores, S. Nadal and O. Romero, Towards Scalable Data Discovery, in: *EDBT*, OpenProceedings.org, 2021, pp. 433–438.

[18] G. Xiao, D. Calvanese, R. Kontchakov, D. Lembo, A. Poggi, R. Rosati and M. Zakharyaschev, Ontology-Based Data Access: A Survey, in: *IJCAI*, ijcai.org, 2018, pp. 5511–5519.

[19] S. Nadal, A. Abello, O. Romero, S. Vansummeren and P. Vassiliadis, Graph-driven Federated Data Management, *IEEE Transactions on Knowledge and Data Engineering* (2021), 1–1. doi:10.1109/TKDE.2021.3077044.

[20] C. Pinkel, C. Binnig, E. Jiménez-Ruiz, E. Kharlamov, A. Nikolov, A. Schwarte, C. Heupel and T. Kraska, IncMap: A Journey towards Ontology-based Data Integration, in: *BTW*, LNI, Vol. P-265, GI, 2017, pp. 145–164.

[21] E. Jiménez-Ruiz, E. Kharlamov, D. Zheleznyakov, I. Horrocks, C. Pinkel, M.G. Skjæveland, E. Thorstensen and J. Mora, BootOX: Practical Mapping of RDBs to OWL 2, in: *ISWC (2)*, Lecture Notes in Computer Science, Vol. 9367, Springer, 2015, pp. 113–132.

[22] I. Bedini, C.J. Matheus, P.F. Patel-Schneider, A. Boran and B. Nguyen, Transforming XML Schema to OWL Using Patterns, in: *ICSC*, IEEE Computer Society, 2011, pp. 102–109.

[23] C. Tsinaraki and S. Christodoulakis, XS2OWL: A Formal Model and a System for Enabling XML Schema Applications to Interoperate with OWL-DL Domain Knowledge and Semantic Web Tools, in: *DELOS*, Lecture Notes in Computer Science, Vol. 4877, Springer, 2007.

[24] P.T.T. Thuy, Y. Lee and S. Lee, DTD2OWL: automatic transforming XML documents into OWL ontology, in: *ICIS*, ACM International Conference Proceeding Series, Vol. 403, ACM, 2009, pp. 125–131.

[25] K.M. Albarrak and E.H. Sibley, A survey of methods that transform data models into Ontology models, in: *IRI*, IEEE Systems, Man, and Cybernetics Society, 2011, pp. 58–65.

[26] J.F. Sequeda, S.H. Tirmizi, Ó. Corcho and D.P. Miranker, Survey of directly mapping SQL databases to the Semantic Web, *Knowl. Eng. Rev.* **26**(4) (2011), 445–486.

[27] B.E. Idrissi, S. Baïna and K. Baïna, Automatic generation of ontology from data models: A practical evaluation of existing approaches, in: *RCIS*, IEEE, 2013, pp. 1–12.

[28] M. Hacherouf, S.N. Bahloul and C. Cruz, Transforming XML documents to OWL ontologies: A survey, *J. Inf. Sci.* **41**(2) (2015), 242–259.

[29] C. Bizer and R. Cyganiak, D2r server-publishing relational databases on the semantic web, in: *Poster at the 5th international semantic web conference*, Vol. 175, 2006.

[30] L.F. de Medeiros, F. Priyatna and Ó. Corcho, MIRROR: Automatic R2RML Mapping Generation from Relational Databases, in: *ICWE*, Lecture Notes in Computer Science, Vol. 9114, Springer, 2015, pp. 326–343.

[31] C.A. Knoblock, P.A. Szekely, J.L. Ambite, A. Goel, S. Gupta, K. Lerman, M. Muslea, M. Taheriyan and P. Mallick, Semi-automatically Mapping Structured Sources into the Semantic Web, in: *ESWC*, Lecture Notes in Computer Science, Vol. 7295, Springer, 2012.

[32] M. Giese, A. Soylu, G. Vega-Gorgojo, A. Waaler, P. Haase, E. Jiménez-Ruiz, D. Lanti, M. Rezk, G. Xiao, Ö.L. Özçep and R. Rosati, Optique: Zooming in on Big Data, *Computer* **48**(3) (2015), 60–67.

[33] N.F. Noy and M.A. Musen, PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment, in: *AAAI/IAAI*, AAAI Press / The MIT Press, 2000, pp. 450–455.

[34] D.L. McGuinness, R. Fikes, J. Rice and S. Wilder, The Chimaera Ontology Environment, in: *AAAI/IAAI*, AAAI Press / The MIT Press, 2000, pp. 1123–1124.

[35] G. Stumme and A. Maedche, FCA-MERGE: Bottom-Up Merging of Ontologies, in: *IJCAI*, Morgan Kaufmann, 2001, pp. 225–234.

[36] P. Mitra, G. Wiederhold and S. Decker, A Scalable Framework for the Interoperation of Information Sources, in: *SWWS*, 2001, pp. 317–329.

[37] D. Dou, D.V. McDermott and P. Qi, Ontology Translation on the Semantic Web, *J. Data Semant.* **2** (2005), 35–57.

[38] S. Raunich and E. Rahm, Target-driven merging of taxonomies with Atom, *Inf. Syst.* **42** (2014), 1–14.

[39] S. Babalou and B. König-Ries, Towards Building Knowledge by Merging Multiple Ontologies with CoMerger: A Partitioning-based Approach, *CoRR* **abs/2005.02659** (2020).

[40] B. Vidé, J. Marty, F. Ravat and M. Chevalier, Designing a Business View of Enterprise Data: An approach based on a Decentralised Enterprise Knowledge Graph, in: *IDEAS*, ACM, 2021, pp. 184–193.

[41] I. Osman, S.B. Yahia and G. Diallo, Ontology Integration: Approaches and Challenging Issues, *Inf. Fusion* **71** (2021), 38–63.

[42] S. Babalou, E. Grygorova and B. König-Ries, What to Do When the Users of an Ontology Merging System Want the Impossible? Towards Determining Compatibility of Generic Merge Requirements, in: *EKAW*, Lecture Notes in Computer Science, Vol. 12387, Springer, 2020.

[43] S. Nadal, A. Abelló, O. Romero, S. Vansummeren and P. Vassiliadis, MDM: Governing Evolution in Big Data Ecosystems, in: *EDBT*, OpenProceedings.org, 2018, pp. 682–685.

[44] D. Calvanese, B. Cogrel, S. Komla-Ebri, R. Kontchakov, D. Lanti, M. Rezk, M. Rodriguez-Muro and G. Xiao, Ontop: Answering SPARQL queries over relational databases, *Semantic Web* **8**(3) (2017), 471–487.

[45] M. Solanki, C. Mader, H. Nagy, M. Mückstein, M. Hanfi, R. David and A. Koller, Ontology-Driven Unified Governance in Software Engineering: The PoolParty Case Study, in: *ESWC (2)*, Lecture Notes in Computer Science, Vol. 10250, 2017, pp. 109–124.

[46] C. Civili, M. Console, G.D. Giacomo, D. Lembo, M. Lenzerini, L. Lepore, R. Mancini, A. Poggi, R. Rosati, M. Ruzzi, V. Santarelli and D.F. Savo, MASTRO STUDIO: Managing Ontology-Based Data Access applications, *Proc. VLDB Endow.* **6**(12) (2013), 1314–1317.

[47] E. Kharlamov, S. Brandt, M. Giese, E. Jiménez-Ruiz, S. Lamparter, C. Neuenstadt, Ö.L. Özçep, C. Pinkel, A. Soylu, D. Zheleznyakov, M. Roshchin, S. Watson and I. Horrocks, Semantic Access to Siemens Streaming Data: the Optique Way, in: *ISWC (Posters & Demos)*, CEUR Workshop Proceedings, Vol. 1486, CEUR-WS.org, 2015.

[48] K.M. Endris, P.D. Rohde, M. Vidal and S. Auer, Ontario: Federated Query Processing Against a Semantic Data Lake, in: *DEXA (1)*, Lecture Notes in Computer Science, Vol. 11706, Springer, 2019, pp. 379–395.

[49] M. Buron, F. Goasdoué, I. Manolescu and M. Mugnier, Obi-Wan: Ontology-Based RDF Integration of Heterogeneous Data, *Proc. VLDB Endow.* **13**(12) (2020), 2933–2936.

[50] F. Priyatna, Ó. Corcho and J.F. Sequeda, Formalisation and experiences of R2RML-based SPARQL to SQL query translation using morph, in: *WWW*, ACM, 2014, pp. 479–490.

[51] M.N. Mami, D. Graux, S. Scerri, H. Jabeen, S. Auer and J. Lehmann, Squerall: Virtual Ontology-Based Access to Heterogeneous and Large Data Sources, in: *ISWC (2)*, Lecture Notes in Computer Science, Vol. 11779, Springer, 2019, pp. 229–245.

[52] A. Bonifati, G.H.L. Fletcher, H. Voigt and N. Yakovets, *Querying Graphs*, Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2018.

[53] J.L.C. Izquierdo and J. Cabot, Discovering Implicit Schemas in JSON Data, in: *ICWE*, Lecture Notes in Computer Science, Vol. 7977, Springer, 2013, pp. 68–83.

[54] E. Jiménez-Ruiz and B.C. Grau, LogMap: Logic-Based and Scalable Ontology Matching, in: *ISWC (1)*, Lecture Notes in Computer Science, Vol. 7031, Springer, 2011, pp. 273–288.

[55] S. Nadal, K. Rabbani, O. Romero and S. Tadesse, ODIN: A Dataspace Management System, in: *ISWC (Satellites)*, CEUR Workshop Proceedings, Vol. 2456, CEUR-WS.org, 2019, pp. 185–188.

[56] F. Priyatna, R. Alonso-Calvo, S. Paraiso-Medina, G. Padron-Sanchez and Ó. Corcho, R2RML-based Access and Querying to Relational Clinical Data with Morph-RDB, in: *SWAT4LS*, CEUR Workshop Proceedings, Vol. 1546, CEUR-WS.org, 2015, pp. 142–151.

[57] E. Iglesias, S. Jozashoori, D. Chaves-Fraga, D. Collarana and M. Vidal, SDM-RDFizer: An RML Interpreter for the Efficient Creation of RDF Knowledge Graphs, in: *CIKM*, ACM, 2020, pp. 3039–3046.

[58] G. Haesendonck, W. Maroy, P. Heyvaert, R. Verborgh and A. Dimou, Parallel RDF generation from heterogeneous big data, in: *SBD@SIGMOD*, ACM, 2019, pp. 1:1–1:6.

[59] M. da Conceição Moraes Batista and A.C. Salgado, Information Quality Measurement in Data Integration Schemas, in: *QDB*, 2007, pp. 61–72.

Appendix A. JSON metamodel constraints

In this appendix, we present the constraints considered for the metamodel we adopt to represent the schemata of JSON datasets (i.e., $\mathcal{M}_{\text{JSON}}$), which is depicted in Section 4.1. Hereinafter, we assume all constraints are applied over a graph G which is typed with respect to $\mathcal{M}_{\text{JSON}}$.

IS-A relationships and constraints on generalizations. Rule 1 restricts instances of J:DataType to be instances of either J:Primitive, J:Object or J:Array. Then, Rules 2, 3, and 4 constrain that for any of such subclass instances, the instantiation of the superclass J:DataType also exists in G. Finally, Rules 5, 6 and 7 determine that the subclasses of J:DataType are disjoint.

$$\forall d\big(\langle d, \texttt{rdf:type}, \texttt{J:DataType}\rangle(G)\big) \Rightarrow \exists d'\big(\langle d', \texttt{rdf:type}, \texttt{J:Primitive}\rangle \vee \langle d', \texttt{rdf:type}, \texttt{J:Object}\rangle \vee \langle d', \texttt{rdf:type}, \texttt{J:Array}\rangle(G)\big) | d = d' \tag{1}$$

$$\forall o\big(\langle o, \texttt{rdf:type}, \texttt{J:Object}\rangle(G)\big) \Rightarrow \exists d\big(\langle d, \texttt{rdf:type}, \texttt{J:DataType}\rangle(G)\big) | o = d \tag{2}$$

$$\forall p\big(\langle p, \texttt{rdf:type}, \texttt{J:Primitive}\rangle(G)\big) \Rightarrow \exists d\big(\langle d, \texttt{rdf:type}, \texttt{J:DataType}\rangle(G)\big) | p = d \tag{3}$$

$$\forall a\big(\langle a, \texttt{rdf:type}, \texttt{J:Array}\rangle(G)\big) \Rightarrow \exists d\big(\langle d, \texttt{rdf:type}, \texttt{J:DataType}\rangle(G)\big) | a = d \tag{4}$$

$$\forall o\big(\langle o, \texttt{rdf:type}, \texttt{J:Object}\rangle(G)\big) \Rightarrow \nexists p\big(\langle p, \texttt{rdf:type}, \texttt{J:Primitive}\rangle(G)\big) | o = p \tag{5}$$

$$\forall o\big(\langle o, \texttt{rdf:type}, \texttt{J:Object}\rangle(G)\big) \Rightarrow \nexists a\big(\langle a, \texttt{rdf:type}, \texttt{J:Array}\rangle(G)\big) | o = a \tag{6}$$

$$\forall p\big(\langle p, \texttt{rdf:type}, \texttt{J:Primitive}\rangle(G)\big) \Rightarrow \nexists a\big(\langle a, \texttt{rdf:type}, \texttt{J:Array}\rangle(G)\big) | p = a \tag{7}$$

Referential integrity constraints. Rule 8 indicates that an edge labeled with J:hasValue will connect instances of either J:Document and J:Object, or instances of J:Key and J:DataType. Similarly, Rule 9 applies the same strategy to connect instances of J:Object and J:Key using edges labeled J:hasKey.

$$\forall d, o\big(\langle d, \texttt{J:hasValue}, o\rangle(G)\big) \Rightarrow \exists d', o'\Big(\big(\langle d', \texttt{rdf:type}, \texttt{J:Document}\rangle(G) \wedge \langle o', \texttt{rdf:type}, \texttt{J:Object}\rangle(G)\big) \vee \big(\langle d', \texttt{rdf:type}, \texttt{J:Key}\rangle(G) \wedge \langle o', \texttt{rdf:type}, \texttt{J:DataType}\rangle(G)\big)\Big) | d = d' \wedge o = o' \tag{8}$$

$$\forall o, k\big(\langle o, \texttt{J:hasKey}, k\rangle(G)\big) \Rightarrow \exists o', k'\big(\langle o', \texttt{rdf:type}, \texttt{J:Object}\rangle \wedge \langle k', \texttt{rdf:type}, \texttt{J:Key}\rangle(G)\big) | o = o' \wedge k = k' \tag{9}$$

Cardinality constraints. Rule 10 states that an instance of J:Document has a single instance of J:Object as root. Then, Rule 11 models a many-to-one relationship between instances of J:Key and J:DataType.

$$\forall d, o\big(\langle d, \texttt{J:hasValue}, o\rangle(G)\big) \Rightarrow \nexists d', o', o''\big(\langle d', \texttt{J:hasValue}, o'\rangle(G) \wedge \langle d', \texttt{J:hasValue}, o''\rangle(G)\big) | d = d' \wedge o = o' \wedge o = o'' \tag{10}$$

$$\forall k, d\big(\langle k, \texttt{J:hasValue}, d\rangle(G)\big) \Rightarrow \nexists k', d'\big(\langle k', \texttt{J:hasValue}, d'\rangle(G) | k = k' \wedge d = d' \tag{11}$$

Appendix B. RDFS metamodel constraints

In this appendix, we present the constraints considered for the fragment of RDFS that we consider in this paper (i.e., $\mathcal{M}_{\text{RDFS}}$), which is depicted in Section 4.2. All such constraints are based on the *RDF Schema 1.1* standard[16]. Hereinafter, we assume all constraints are applied over a graph G which is typed with respect to $\mathcal{M}_{\text{RDFS}}$.

IS-A relationships and constraints on generalizations. Rule 12 restricts instances of rdfs:Resource to be instances of either rdf:Property or rdfs:Class. Then, Rules 13 and 14, constrain that for any of such subclass instances, the instantiation of the superclass rdfs:Resource also exists in G. Next, Rule 15 denote that instances of rdfs:Datatype are also instances of rdfs:Class.

$$\forall r\big(\langle r, \texttt{rdf:type}, \texttt{rdfs:Resource}\rangle(G)\big) \Rightarrow \exists r'\big(\langle r', \texttt{rdf:type}, \texttt{rdf:Property}\rangle(G) \vee \langle r', \texttt{rdf:type}, \texttt{rdfs:Class}\rangle(G)\big) | r = r' \tag{12}$$

[16] https://www.w3.org/TR/rdf-schema/

$$\forall p\big(\langle p, \texttt{rdf:type}, \texttt{rdf:Property}\rangle(G)\big) \Rightarrow \exists r\big(\langle r, \texttt{rdf:type}, \texttt{rdfs:Resource}\rangle(G)\big)|p = r \tag{13}$$

$$\forall c\big(\langle c, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle(G)\big) \Rightarrow \exists r\big(\langle r, \texttt{rdf:type}, \texttt{rdfs:Resource}\rangle(G)\big)|c = r \tag{14}$$

$$\forall d\big(\langle d, \texttt{rdf:type}, \texttt{rdfs:Datatype}\rangle(G)\big) \Rightarrow \exists c\big(\langle c, \texttt{rdf:type}, \texttt{rdfs:Class}\rangle(G)\big)|d = c \tag{15}$$

$$\tag{16}$$

Referential integrity constraints. Rule 17 indicates that the rdfs:domain of an instance of rdf:Property is an instance of rdfs:Class. Similarly, Rule 18, constraints that the rdfs:range of instances of rdf:Property are also instances of rdfs:Class.

$$\begin{gathered}\forall p, c\big(\langle p, \texttt{rdfs:domain}, c\rangle(G)\big) \Rightarrow \exists p', c'\big(\langle p', \texttt{rdf:type}, \texttt{rdf:Property}\rangle(G) \wedge \\ \langle c', \texttt{rdf:type}, \texttt{rdfs:Class}\rangle(G)\big)|p = p' \wedge c = c'\end{gathered} \tag{17}$$

$$\begin{gathered}\forall p, c\big(\langle p, \texttt{rdfs:range}, c\rangle(G)\big) \Rightarrow \exists p', c'\big(\langle p', \texttt{rdf:type}, \texttt{rdf:Property}\rangle(G) \wedge \\ \langle c', \texttt{rdf:type}, \texttt{rdfs:Class}\rangle(G)\big)|p = p' \wedge c = c'\end{gathered} \tag{18}$$

B.1. Schema integration constraints

Here we present the constraints for the RDFS metamodel extension we consider to support the annotated integration process depicted in Section 5.

IS-A relationships. Rule 19 states that any instance of :IntegratedResource is also an instance of rdfs:Resource. Similarly, Rule 20, denotes that any instance of :JoinProperty is also an instance of rdf:Property.

$$\forall p\big(\langle p, \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle(G)\big) \Rightarrow \exists r\big(\langle r, \texttt{rdf:type}, \texttt{rdfs:Resource}\rangle(G)\big)|p = r \tag{19}$$

$$\forall p\big(\langle p, \texttt{rdf:type}, \texttt{:JoinProperty}\rangle(G)\big) \Rightarrow \exists r\big(\langle r, \texttt{rdf:type}, \texttt{rdf:Property}\rangle(G)\big)|p = r \tag{20}$$

Referential integrity constraints. Rule 21 denotes that the rdfs:domain of a :JoinProperty instance is an :IntegratedResource.

$$\begin{gathered}\forall p, c\big(\langle p, \texttt{rdfs:domain}, c\rangle(G)\big) \Rightarrow \exists p', c'\big(\langle p', \texttt{rdf:type}, \texttt{:JoinProperty}\rangle(G) \wedge \\ \langle c', \texttt{rdf:type}, \texttt{:IntegratedResource}\rangle(G)\big)|p = p' \wedge c = c'\end{gathered} \tag{21}$$