An Integration-Oriented Ontology to Govern Evolution in Big Data Ecosystems

Sergi Nadal[a,*], Oscar Romero[a], Alberto Abelló[a], Panos Vassiliadis[b], Stijn Vansummeren[c]

[a] *Universitat Politècnica de Catalunya - BarcelonaTech*
[b] *University of Ioannina*
[c] *Université Libre de Bruxelles*

Abstract

Big Data architectures allow to flexibly store and process heterogeneous data, from multiple sources, in their original format. The structure of those data, commonly supplied by means of REST APIs, is continuously evolving. Thus data analysts need to adapt their analytical processes after each API release. This gets more challenging when performing an integrated or historical analysis. To cope with such complexity, in this paper, we present the Big Data Integration ontology, the core construct to govern the data integration process under schema evolution by systematically annotating it with information regarding the schema of the sources. We present a query rewriting algorithm that, using the annotated ontology, converts queries posed over the ontology to queries over the sources. To cope with syntactic evolution in the sources, we present an algorithm that semi-automatically adapts the ontology upon new releases. This guarantees ontology-mediated queries to correctly retrieve data from the most recent schema version as well as correctness in historical queries. A functional and performance evaluation on real-world APIs is performed to validate our approach.

Keywords: Data integration, Evolution, Semantic web

1. Introduction

Big Data ecosystems enable organizations to evolve their decision making processes from classic stationary data analysis [1] (e.g., transactional) to situational data analysis [15] (e.g., social networks). Situational data are commonly obtained in the form of data streams supplied by third party data providers (e.g., Twitter or Facebook), by means of web services (or APIs). Those APIs offer a

*Corresponding author. Address: Campus Nord Omega-125, UPC - dept ESSI, C/Jordi Girona Salgado 1-3, 08034 Barcelona, Spain

Email addresses: `snadal@essi.upc.edu` (Sergi Nadal), `oromero@essi.upc.edu` (Oscar Romero), `aabello@essi.upc.edu` (Alberto Abelló), `pvassil@cs.uoi.gr` (Panos Vassiliadis), `stijn.vansummeren@ulb.ac.be` (Stijn Vansummeren)

part of their data ecosystem at a certain price allowing external data analysts to enrich their data pipelines with them. With the rise of the RESTful architectural style for web services [22], providers have flexible mechanisms to share such data, usually semi-structured (i.e., JSON), over web protocols (e.g., HTTP). However, such flexibility can be often a disadvantage for analysts. In contrast to other protocols offering machine-readable contracts for the structure of the provided data (e.g., SOAP), web services using REST typically do not publish such information. Hence, *analysts need to go over the tedious task of carefully studying the documentation and adapting their processes to the particular schema provided*. Besides the aforementioned complexity imposed by REST APIs, there is a second challenge for data analysts. *Data providers are constantly evolving such endpoints*[1,2], hence *analysts need to continuously adapt the dependent processes to such changes*. Previous work on schema evolution has focused on software obtaining data from relational views [17, 24]. Such approaches rely on the capacity to veto changes affecting consumer applications. Those techniques are not valid in our setting, due to the lack of explicit schema information and the impossibility to prevent changes from third party data providers.

Given this setting, the problem is how to aid the data analyst in the presence of schema changes by (a) understanding what parts of the data structure change and (b) adapting her code to this change.

Providing an integrated view over an evolving and heterogeneous set of data sources is a challenging problem, commonly referred as the data variety challenge [8], that traditional data integration techniques fail to address. An approach to tackle it is to leverage on Semantic Web technologies, and the so-called ontology-based data access (OBDA). OBDA are a class of systems that enable end-users to query an integrated set of heterogeneous and disparate data sources decreasing the need for IT support [23]. OBDA achieves its purpose by providing a conceptualization of the domain of interest, via an ontology, allowing users to pose ontology-mediated queries (OMQs), and thus creating a separation of concerns between the conceptual and the database level. Due to the simplicity and flexibility of ontologies, they constitute an ideal tool to model such heterogeneous environments. However, such flexibility is also one of its biggest drawbacks, as OBDA currently has no means to provide continuous adaptation to changes in the sources (e.g., schema evolution), and thus causing queries to crash.

The problem is not straightforwardly addressable, as current OBDA approaches, which are built upon generic reasoning in description logics (DLs), represent schema mappings following the *global-as-view* (GAV) approach [12]. In GAV, elements of the ontology are characterized in terms of a query over the source schemata. This provides simplicity in the query answering methods, which consists of unfolding the queries to the sources. Changes in the source schemata, however, will invalidate the mappings. In contrast, *local-as-view* (LAV) schema

[1] https://dev.twitter.com/ads/overview/recent-changes
[2] https://developers.facebook.com/docs/apps/changelog

mappings characterize elements of the source schemata in terms of a query over the ontology. They are naturally suited to accomodate dynamic environments, as we will see. The trade-off however, comes at the expense of query answering, which becomes a computationally complex task that might require reasoning [9]. To this end, we aim to bridge this gap by providing a new approach to OBDA with LAV mapping assertions, while maintaining query answering tractable. We follow a vocabulary-based approach which rely on tailored metadata models to design the ontology (i.e., a set of design guidelines). This allows to annotate the data integration constructs with semantic annotations, enabling to automate the process of evolution and resolve query answering without ambiguity. Oppositely to reasoning-based approaches, vocabulary-based OBDA is not limited by the expressiveness of a concrete DL for query answering, as it does not rely on generic reasoning techniques but on ad-hoc algorithms that leverage such semantic annotations.

Our approach builds upon the well-known framework for data integration [12], and it is divided in two levels represented by graphs (i.e., Global and Source graphs) in order to provide analysts with an integrated and format-agnostic view of the sources. By relying on wrappers (from the well-known mediator/wrapper architecture for data integration [7]) we are able to accomodate different kinds of data sources, as the query complexity is delegated to wrappers and the ontology is only concerned with how to join them and what attributes are projected. Additionally, we allow the ontology to contain elements that do not exist in the sources (i.e., syntactic sugar for data analysts), such as taxonomies, to facilitate querying. The process of query answering is reduced to properly resolving the LAV mapping assertions, relying on the annotated ontology, in order to construct an expression fetching the attributes provided by the wrappers. Finally, we exploit this structure to handle the evolution of source schema via semi-automated transformations on the ontology upon service releases.

Contributions. The main contributions of this paper are as follows:

- We introduce a structured ontology based on an RDF vocabulary that allows to model and integrate evolving data from multiple providers. As an add-on, we take advantage of RDF's nature to semantically annotate the data integration process.

- We provide a method that handles schema evolution on the sources. According to our industry applicability study, we flexibly accommodate source changes by only applying changes to the ontology, dismissing the need to change the analyst's queries.

- We present a query answering algorithm that using the annotated elements in the ontology is capable of unambiguously resolving LAV mappings. Given a OMQ over the ontology, we are capable of manipulating it yielding an equivalent query over the sources. We further provide a theoretical and practical study of its complexity and limitations.

- We assess our method by performing a functional and performance evaluation. The former reveals that our approach is capable of semi-automatically accomodating all structural changes concerning data ingestion, which on average makes up 71.62% of the changes occurring on widely used APIs.

Outline. The rest of the paper is structured as follows. Section 2 describes a running example and formalizes the problem at hand. Section 3 discusses the constructs of the Big Data Integration ontology and its RDF representation. Section 4 introduces the techniques to manage schema evolution. Section 5 presents the query answering algorithm. Section 6 reports on the evaluation results. Sections 7 and 8 discuss related work and conclude the paper, respectively.

2. Overview

Our approach (see Figure 1) relies on a two-level ontology of RDF named graphs to accomodate schema evolution in the data sources. Such graphs are built based on a RDF vocabulary tailored for data integration. Precisely, we divide it into the *Global graph* ($\mathcal{G}$), and the *Source graph* ($\mathcal{S}$). Briefly, $\mathcal{G}$ represents an integrated view of the domain of interest (also known as domain ontology), while $\mathcal{S}$ represents data sources, wrappers and their schemata. On the one hand, data analysts issue OMQs to $\mathcal{G}$. We also assume a triplestore with a SPARQL endpoint supporting the RDFS entailment regime (e.g., subclass relations are automatically inferred) [26]. On the other hand, we have a set of data sources, each with a set of wrappers querying it. Different wrappers for a data source represent different schema versions. Under the assumption that wrappers provide a flat structure in first normal form, we can easily depict an accurate representation of their schema into $\mathcal{S}$. To acommodate a LAV approach, each wrapper in $\mathcal{S}$ is related to the fragment of $\mathcal{G}$ for which it provides data.

The management of such a complex structure (i.e., modifying it upon schema evolution in the sources) is a hard task to automate. To this end, we introduce the role of data steward as an analogy to the database administrator in traditional relational settings. Aided by semi-automatic techniques, s/he is responsible for (a) registering the wrappers of newly incoming, or evolved, data sources in $\mathcal{S}$, and (b) make such data available to analysts by defining LAV mappings to $\mathcal{G}$ (i.e., enriching the ontology with the mapping representations). With such setting, intuitively the problem consists of given a query over $\mathcal{G}$, to derive an equivalent query over the wrappers leveraging on $\mathcal{S}$. Throughout the rest of this section, we introduce the running example and the formalism behind our approach. To make a clear distinction among concepts, hereinafter, we will use *italics* to refer to elements in $\mathcal{G}$, while $\mathsf{sans\ serif}$ font to refer to elements in $\mathcal{S}$. Additionally, to refer to RDF constructs, we will use `typewriter` font.

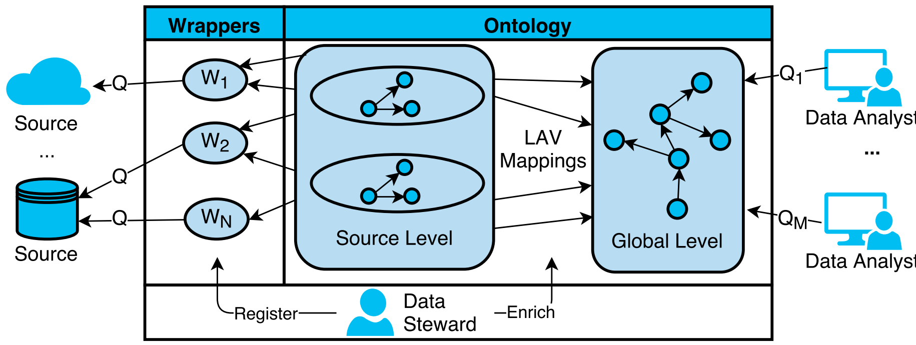

Figure 1: High-level overview of our approach

2.1. Running Example

As an exemplary use case we take the H2020 SUPERSEDE project[3]. It aims to support decision-making in the evolution and adaptation of software services and applications (i.e., *SoftwareApps*) by exploiting end-user feedback and monitored runtime data, with the overall goal of improving end-users' quality of experience. For the sake of this case study, we narrow the scope to video on demand (VoD) monitored data (i.e., *Monitor* tools generating *InfoMonitor* events) and textual feedback from social networks such as Twitter (i.e., *FeedbackGathering* tools generating *UserFeedback* events). This scenario is conceptualized in the UML depicted in Figure 2, which we use as a starting point to provide a high-level representation of the domain of interest that is later used to generate the ontological knowledge captured in $\mathcal{G}$. Figure 3 in Section 3 depicts the RDF-based representation of the UML diagram used in our approach, which we will introduce in detail in that section.

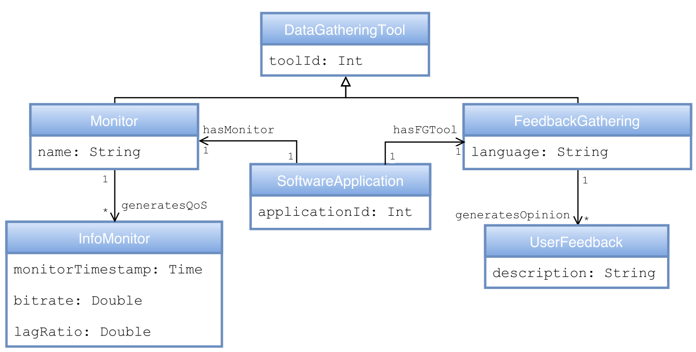

Figure 2: UML conceptual model for the SUPERSEDE case study

Next, let us assume three data sources, in the form of REST APIs, and re-

[3] https://www.supersede.eu

spectively one wrapper querying each. The first data source provides information related to the VoD monitor, which consist of JSON documents as depicted in Code 1. We additionally define a wrapper on top of it obtaining the monitorId of the monitor and computing the lag ratio metric (a quality of service measure computed as the fraction of wait and watch time) indicating the percentage of time a user is waiting for a video. The query of this wrapper is depicted in Code 2 using MongoDB syntax[4], where for each tuple the attribute VoDmonitorId (renamed from monitorId in the JSON) and lagRatio are projected (respectively mapping to the conceptual attributes *toolId* and *lagRatio*).

```
{
  "monitorId": 12,
  "timestamp": 1475010424,
  "bitrate": 6,
  "waitTime": 3,
  "watchTime": 4
}
```

Code 1: Sample JSON for VoD monitors

```
db.getCollection("vod").aggregate([
  {$project: {
    "VoDmonitorId":"$monitorId",
    "lagRatio": {$divide : ["$waitTime","
        $watchTime"]}}
  }
])
```

Code 2: Wrapper projecting attributes VoDmonitorId and lagRatio (using MongoDB's Aggregation Framework syntax)

For the sake of simplicity, hereinafter, we will represent wrappers as relations where their schema are the attributes projected by the queries, dismissing the details of the underlying query. Hence, the previous wrapper would be depicted as $w_1(\mathsf{VoDmonitorId}, \mathsf{lagRatio})$ (note that the JSON key monitorId has been renamed to VoDmonitorId). To complete our running example, we define a wrapper $w_2(\mathsf{FGId}, \mathsf{tweet})$ providing, respectively, the *toolId* for the *FeedbackGathering* at hand and the *description* for such *UserFeedback*. Finally, the wrapper $w_3(\mathsf{TargetApp}, \mathsf{MonitorId}, \mathsf{FeedbackId})$ states for each *SoftwareApplication* the *toolId* of its associated *Monitor* and *FeedbackGathering* tools. Table 1 depicts an example of the output generated by each wrapper.

w_1

VoDmonitorId	lagRatio
12	0.75
12	0.90
18	0.1

w_2

FGId	tweet
77	"I continuously see the loading symbol"
45	"Your video player is great!"

w_3

TargetApp	MonitorId	FeedbackId
1	12	77
2	18	45

Table 1: Sample output for each of the exemplary wrappers.

Now, the goal is to enable data analysts to query the attributes of the ontology-based representation of the UML diagram (i.e., $\mathcal{G}$) by navigating over

[4]Note that the use of the `aggregate` keyword is used to invoke the aggregate querying framework. The `aggregate` keyword does not entail grouping unless the `$group` keyword is used. Thus, note no aggregation is performed in this query.

the classes, such that the sources are automatically accessed. Throughout the paper we will make use of the exemplary query retrieving for each *applicationId* its *lagRatio* instances. Hence, the task consists of rewriting such OMQ to an equivalent one over the wrappers, which can be translated to the following relational algebra expression: $\Pi_{w_3.\mathsf{TargetApp},w_1.\mathsf{lagRatio}}(w_1 \underset{\mathsf{VoDmonitorId}=\mathsf{MonitorId}}{\bowtie} w_3)$. Table 2 depicts an example of the output generated by such query.

TargetApp	lagRatio
1	0.75
1	0.90
2	0.1

Table 2: Sample output for the exemplary query.

Assume now that the first data source releases a new version of its API and in the new schema lagRatio has been renamed to bufferingRatio. Hence, a new wrapper $w_4(\mathsf{VoDmonitorId},\mathsf{bufferingRatio})$ is defined. With such setting, the analyst should not be aware of such schema evolution, but now the query should consider both versions and be automatically rewritten to the following expression: $\Pi_{w_3.\mathsf{TargetApp},w_1.\mathsf{lagRatio}}(w_1 \underset{\mathsf{VoDmonitorId}=\mathsf{MonitorId}}{\bowtie} w_3) \bigcup \Pi_{w_3.\mathsf{TargetApp},w_4.\mathsf{bufferingRatio}}(w_4 \underset{\mathsf{VoDmonitorId}=\mathsf{MonitorId}}{\bowtie} w_3)$.

2.2. Notation

We consider a set of data sources $D = \{D_1, \ldots, D_n\}$, where each D_i consists of a set of wrappers $\{w_1, \ldots, w_m\}$ representing views over different schema versions. We define the operator $source(w)$, which returns the data source D to which w belongs to. As previously stated, a wrapper is represented as a relation with the attributes its query projects. We distinguish between ID and non-ID attributes, hence a wrapper is defined as $w(\overline{a_{ID}}, \overline{a_{nID}})$, where $\overline{a_{ID}}$ and $\overline{a_{nID}}$ are respectively the set of its ID attributes and non-ID attributes.

Example. The VoD monitoring API would be depicted as $D_1 = \{w_1(\{\mathsf{VoDmonitorId}\}, \{\mathsf{lagRatio}\}), w_4(\{\mathsf{VoDmonitorId}\}, \{\mathsf{bufferingRatio}\})\}$, the feedback gathering API as $D_2 = \{w_2(\{\mathsf{FGId}\}, \{\mathsf{tweet}\})$ and the relationship API as $D_3 = \{w_3(\{\mathsf{TargetApp}, \mathsf{MonitorId}, \mathsf{FeedbackId}\}, \{\})$.

Wrappers can be joined to each other by means of a restricted equi-join on IDs ($\widetilde{\bowtie}$). The semantics of $\widetilde{\bowtie}$ are those of an equi-join ($w_i \underset{a=b}{\bowtie} w_j$), but only valid if $a \in w_i.\overline{a_{ID}}$ and $b \in w_j.\overline{a_{ID}}$. We also define the projection operator $\widetilde{\Pi}$, whose semantics are likewise a standard projection for non-ID attributes. We do not permit to project out any ID attribute, as they are necessary for $\widetilde{\bowtie}$. With such constructs, we can now define the concept of a walk over the wrappers (W), which consists of a relational algebra expression where wrappers are joined ($\widetilde{\bowtie}$) and their attributes are projected ($\widetilde{\Pi}$). Thus, we formally define a walk as $W = \widetilde{\Pi}(w_1)\widetilde{\bowtie}\ldots\widetilde{\bowtie}\widetilde{\Pi}(w_k)$. Furthermore, we work under the assumption that schema versions from the same data source should not be joined (e.g., w_1 and w_4 in the running example). To formalize this assumption let $wrappers(W)$ denote the set of wrappers used in walk W. Then we require that $\forall w_i, w_j \in wrappers(W) : source(w_i) \neq source(w_j)$. Note that a walk can also be seen as a conjunctive query over the wrappers (i.e., select-project-join expression), thus two walks are equivalent if they join the same wrappers dismissing the order how this is done. Consider, however, that as the operator $\widetilde{\Pi}$ does not project out ID attributes, all ID attributes will be part of the output schema.

Example. The exemplary query (i.e., for each *applicationId* fetch its *lagRatio* instances) would consist of two walks $W_1 = \widetilde{\Pi}_{\mathsf{lagRatio}}(w_1) \underset{\mathsf{VoDmonitorId=MonitorId}}{\widetilde{\bowtie}} \widetilde{\Pi}_{\mathsf{TargetApp}}(w_3)$ and $W_2 = \widetilde{\Pi}_{\mathsf{bufferingRatio}}(w_4) \underset{\mathsf{VoDmonitorId=MonitorId}}{\widetilde{\bowtie}} \widetilde{\Pi}_{\mathsf{TargetApp}}(w_3)$.

Next, we formalize the ontology $\mathcal{T}$ as a 3-tuple $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ of RDF named graphs. The Global graph ($\mathcal{G}$) contains the concepts and relationships that analysts will use to query, the source graph ($\mathcal{S}$) the data sources and the schemata of wrappers, and the mappings graph ($\mathcal{M}$) the LAV mappings between $\mathcal{S}$ and $\mathcal{G}$. Recall that data analysts pose OMQs over $\mathcal{G}$, however we do not allow arbitrary queries. We restrict OMQs to a subset of standard SPARQL defining subgraph patterns of $\mathcal{G}$, and only project elements of such pattern. Code 3 depicts the template of the permitted queries. Precisely, $attr_1, \ldots, attr_n$ must be attribute URIs (i.e., mapping to the UML attributes in Fig. 2), where each $attr_i$ has an invited variable $?v_i$ in the SELECT clause. The set of triples in the WHERE clause must define a connected subgraph of $\mathcal{G}$. On the one hand, it contains triples of the form $\langle s_i, hasFeature, attr_i \rangle$, where s_i are class URIs (i.e., mapping to UML classes) and *hasFeature* a predicate stating that $attr_i$ is attribute of class s_i. On the other hand, it contains triples of the form $\langle s_j, p_j, o_j \rangle$, where s_j and o_j are class URIs (i.e., mapping to UML classes) and p_i predicate URIs (i.e., mapping to relationships between UML classes).

```
SELECT ?v1 ... ?vn
FROM G
WHERE {
 VALUES (?v1 ... ?vn) { (attr1 ... attrn) }
 s1 p1 attr1 .
 ...
 sn pn attrn .
 ...
 sm pm om
}
```

Code 3: Template for accepted SPARQL queries

OMQs are meant to be translated to sets of walks, to this end the aforementioned SPARQL queries must be parsed and manipulated. This task can be simplified leveraging on SPARQL Algebra[5], where the semantics of the query evaluation are specified. Libraries such as ARQ[6] provide mechanisms to get such algebraic structure for a given SPARQL query. Code 4 depicts the algebra structure generated after parsing the subset of permitted SPARQL queries.

```
(project (?v_1 ... ?v_n)
  (join
    (table (vars ?v_1 ... ?v_n)
      (row [?v_1 attr_1] ... [?v_n attr_n])
    )
    (bgp
      (triple s_1 p_1 attr_1)
      ...
      (triple s_n p_n attr_n)
      ...
      (triple s_m p_m o_m)
)))))
```

Code 4: SPARQL algebra for the accepted SPARQL queries

In order to easily manipulate such algebraic structures, we formalize the allowed SPARQL queries as $Q_{\mathcal{G}} = \langle \pi, \varphi \rangle$, where π is the set of projected attributes (i.e., the URIs $attr_1$, ..., $attr_n$) and φ the graph pattern specified under the `bgp` clause (i.e., basic graph pattern). Note that $\pi \subseteq V(\varphi)$, where $V(\varphi)$ returns the vertex set of φ.

Example. The exemplary query is depicted using SPARQL in Code 5. Alternatively, it would be represented as $\pi = \{lagRatio, applicationId\}$, and φ the subgraph $applicationId \xleftarrow[hasFeature]{} SoftwareApplication \xrightarrow[hasMonitor]{} Monitor \xrightarrow[generatesQoS]{} InfoMonitor \xrightarrow[hasFeature]{} lagRatio$.

```
SELECT ?x ?y
FROM G
WHERE {
 VALUES (?x ?y) { (applicationId lagRatio) }
 SoftwareApplication hasFeature applicationId .
 SoftwareApplication hasMonitor Monitor .
 Monitor generatesQoS InfoMonitor .
 InfoMonitor hasFeature lagRatio
}
```

Code 5: Running example's SPARQL query

The wrappers and the ontology are linked by means of schema mappings. Those are commonly formalized using tuple-generating dependencies (tgds) [5], which are logical expressions of the form $\forall x(\exists y \Phi(x, y) \mapsto \exists z \Psi(x, z))$, where

[5] https://www.w3.org/2001/sw/DataAccess/rq23/rq24-algebra.html
[6] https://www.w3.org/2011/09/SparqlAlgebra/ARQalgebra

Φ and Ψ are conjunctive queries. However, in our context we serialize such mappings in the graph $\mathcal{M}$, and not as separated logical expressions. Hence, we define a LAV mapping for a wrapper w as $LAV(w) : w \mapsto \varphi_{\mathcal{G}}$, where $\varphi_{\mathcal{G}}$ is a subgraph of $\mathcal{G}$. We additionally consider a function $F : a_w \mapsto a_m$, that translates the name of an attribute in $\mathcal{S}$ to its corresponding conceptual representation in $\mathcal{G}$. Such function allows us to denote semantic equivalence between physical and conceptual attributes in the ontology (respectively, in $\mathcal{S}$ and $\mathcal{G}$). Intuitively, F forces a physical attribute in the sources to map to one and only one conceptual feature in $\mathcal{G}$. As schema mappings, this function is also serialized in $\mathcal{M}$.

Example. The LAV mapping for w_1 would be the subgraph $Monitor \xrightarrow[generatesQoS]{} InfoMonitor$ (also including all class attributes). Regarding F, the function would make the conversions $w_1.\mathsf{VoDmonitorId} \mapsto toolId$ and $w_1.\mathsf{lagRatio} \mapsto lagRatio$.

2.3. Problem statement

In order to introduce the problem statement we must first introduce the notions of *coverage* and *minimality* for a query $Q_{\mathcal{G}}$ over $\mathcal{G}$ and a walk W. *Coverage* is formalized as $\bigcup_{w \in wrappers(W)} LAV(w) \supseteq Q_{\mathcal{G}}$, which states that a walk covers the query if the union of the LAV graphs of the wrappers participating in the walk subsume $Q_{\mathcal{G}}$. *Minimality* is formalized as $\forall_{w \in W}(coverage(W, Q_{\mathcal{G}}) \wedge \neg coverage(W \setminus w, Q_{\mathcal{G}}))$, which states that if any wrapper is removed from a covering walk, then the walk is not covering anymore. Intuitively, these properties guarantee that a walk answering a query contains all the required attributes and joins, and each wrapper contributes with at least one attribute.

Now, with the previously introduced formalization and properties, we can state the problem of ontology-based query answering under LAV mappings as a faceted search over the wrappers with the goal of finding all possible ways to obtain the requested attributes. Given an OMQ $Q_{\mathcal{G}}$, we aim at finding a set of non-equivalent walks $\mathcal{W}$ such that each $W \in \mathcal{W}$ is *covering* and *minimal* with respect to $Q_{\mathcal{G}}.\varphi$. As a result, we obtain a union of conjunctive queries, which corresponds to the union of all the covering and minimal walks found for $Q_{\mathcal{G}}.\varphi$.

3. Big Data Integration ontology

In this section, we present the Big Data Integration ontology (BDI), the metadata artifact that enables a systematic approach for the data integration system governance when ingesting and analysing the data. To this end, we have followed the well-known theory on data integration [12] and divided it into two levels (by means of RDF named graphs): the Global and Source graphs, respectively $\mathcal{G}$ and $\mathcal{S}$, linked via mappings $\mathcal{M}$. Thanks to the extensibility of RDF, it further enables us to enrich $\mathcal{G}$ and $\mathcal{S}$ with semantics such as data types. In this section we present the RDF vocabulary to be used to represent $\mathcal{G}$ and $\mathcal{S}$. To do so, we present a metamodel for the global and source ontologies that current models (i.e., $\mathcal{G}$ and $\mathcal{S}$) must mandatorily follow. In the following subsections, we elaborate on each graph and present its RDF representation.

3.1. Global graph

The Global graph $\mathcal{G}$ reflects the main domain concepts, relationships among them and features of analysis (i.e., maps to the role of a UML diagram in a machine-readable format). Its elements are defined in terms of the vocabulary users will use when posing queries. The metadata model for $\mathcal{G}$ distinguishes concepts from features, the former mimicking classes and the latter attributes in a UML diagram. Concepts can be linked by means of domain-specific object properties, which implicitely determine their domain and range. Such properties will be used for data analysts to navigate the graph, dismissing the need of specifying how the underlying sources are joined. The link between a concept and its set of features is represented via `G:hasFeature`. In order to disambiguate the query rewriting process we restrict features to belong to only one concept. Additionally, it is possible to define a taxonomy of features, which will denote related semantic domains (e.g., the feature `sup:monitorId` is subclass of `sc:identifier`). Features can be enriched with new semantics to aid the data management and analysis phases. In this paper, we narrow the scope to data types for features, widely used in data integrity management.

Code 6 provides the triples that compose $\mathcal{G}$ in Turtle RDF notation[7]. It contains the main metaclasses (using the namespace prefix `G`[8] as main namespace) which all features of analysis will instantiate. Concepts and features can reuse existing vocabularies by following the principles of the Linked Data (LD) initiative. Additionally, we include elements for data types on features linked using `G:hasDatatype`, albeit their maintenance is out of the scope of this paper. Following the same LD philosophy, we reuse the `rdfs:Datatype` class to instantiate data types. With such design, we favor the elements of $\mathcal{G}$ to be of any of the available types in XML Schema (prefix `xsd`[9]). Finally, note that we focus on non-complex data types, however our model can be easily extended to include complex types [4].

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix voaf: <http://purl.org/vocommons/voaf#> .
@prefix vann: <http://purl.org/vocab/vann/> .
@prefix G: <http://www.essi.upc.edu/~snadal/BDIOntology/Global/> .

<http://www.essi.upc.edu/~snadal/BDIOntology/Global/> rdf:type voaf:Vocabulary ;
       vann:preferredNamespacePrefix "G";
       vann:preferredNamespaceUri "http://www.essi.upc.edu/~snadal/BDIOntology/Global";
       rdfs:label "The Global graph vocabulary" .

G:Concept rdf:type rdfs:Class;
       rdfs:isDefinedBy <http://www.essi.upc.edu/~snadal/BDIOntology/Global/> .

G:Feature rdf:type rdfs:Class;
       rdfs:isDefinedBy <http://www.essi.upc.edu/~snadal/BDIOntology/Global/> .

G:hasFeature rdf:type rdf:Property ;
```

[7] https://www.w3.org/TR/turtle

[8] http://www.essi.upc.edu/~snadal/BDIOntology/Global

[9] http://www.w3.org/2001/XMLSchema

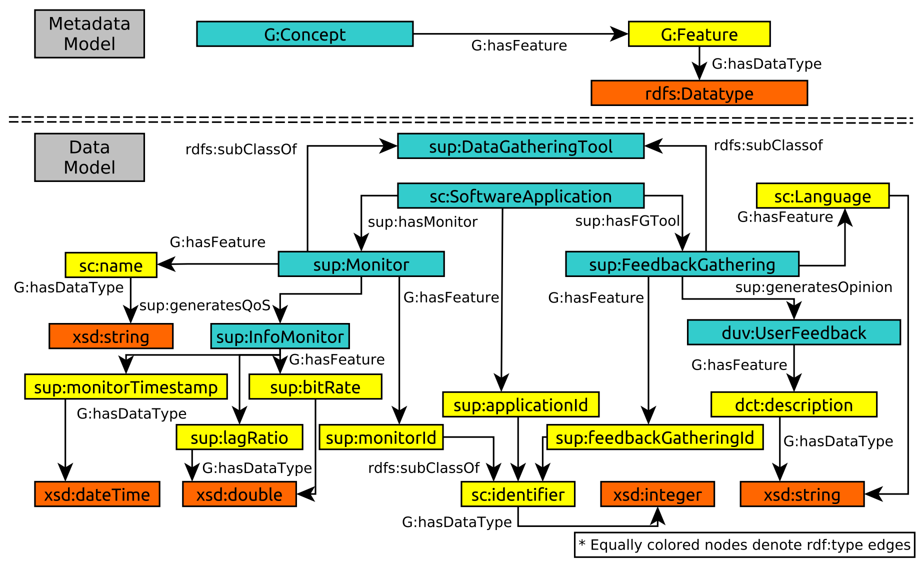

Figure 3: RDF dataset of the metadata model and data model of $\mathcal{G}$ for the SUPERSEDE running example. For interpretation of the references to color in the text, the reader is referred to the web version of this article.

```
        rdfs:isDefinedBy <http://www.essi.upc.edu/~snadal/BDIOntology/Global/> ;
        rdfs:domain G:Concept ;
        rdfs:range G:Feature .

G:hasDataType rdf:type rdf:Property ;
        rdfs:isDefinedBy <http://www.essi.upc.edu/~snadal/BDIOntology/Global/> ;
        rdfs:domain G:Feature ;
        rdfs:range rdfs:Datatype .
```

Code 6: Metadata model for $\mathcal{G}$ in Turtle notation

Example. Figure 3 depicts the instantiation of $\mathcal{G}$ in the SUPERSEDE case study, as presented in the UML diagram in Figure 2 (for the sake of conciseness only a fragment is depicted). The color of the elements represent typing (i.e., `rdf:type` links). Note that, in order to comply with the design constraints of $\mathcal{G}$ (i.e., a feature can only belong to one concept), the *toolId* feature has been explicited and made distinguishable to `sup:monitorId` and `sup:feedbackGatheringId` respectively for classes *Monitor* and *FeedbackGathering*. When possible, vocabularies are reused, namely https://www.w3.org/TR/vocab-duv (prefix `duv`) for feedback elements as well as http://dublincore.org/documents/dcmi-terms (prefix `dct`) or http://schema.org (prefix `sc`). However, when no vocabulary is available we define the custom SUPERSEDE vocabulary (prefix `sup`).

3.2. Source graph

The purpose of the Source graph $\mathcal{S}$ is to model the different wrappers and their provided schema. To this end, we define the metaconcept `S:DataSource` which models the different data sources (e.g., Twitter REST API). In $\mathcal{S}$, we additionally encode the necessary information for schema versioning, hence we define the metaconcept `S:Wrapper` which will model the different schema versions for a data source, which in turn consist of a representation of the projected attributes, modeled in the metaconcept `S:Attribute`. We embrace the reuse of attributes within wrappers of the same data source, as we assume the semantics do not differ across schema versions, however that assumption is not realistic among different data sources (e.g., not necessarily a timestamp has the same meaning in the VoD monitor and the Twitter API). Therefore, we encode in the attribute names the prefix of the data source they correspond to (e.g., for a data source D, its wrappers W and W' respectively provide the attributes $\{\texttt{D/a}, \texttt{D/b}\}$ and $\{\texttt{D/a}, \texttt{D/c}\}$). Code 7 depicts the metadata model for $\mathcal{S}$ in Turtle RDF notation (using prefix `S`[10] as main namespace).

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix voaf: <http://purl.org/vocommons/voaf#> .
@prefix vann: <http://purl.org/vocab/vann/> .
@prefix S: <http://www.essi.upc.edu/~snadal/BDIOntology/Source/> .

<http://www.essi.upc.edu/~snadal/BDIOntology/Source/> rdf:type voaf:Vocabulary ;
       vann:preferredNamespacePrefix "S";
       vann:preferredNamespaceUri "http://www.essi.upc.edu/~snadal/BDIOntology/Source";
       rdfs:label "The Source graph vocabulary" .

S:DataSource rdf:type rdfs:Class;
       rdfs:isDefinedBy <http://www.essi.upc.edu/~snadal/BDIOntology/Source/> .

S:Wrapper rdf:type rdfs:Class;
       rdfs:isDefinedBy <http://www.essi.upc.edu/~snadal/BDIOntology/Source/> .

S:Attribute rdf:type rdfs:Class;
       rdfs:isDefinedBy <http://www.essi.upc.edu/~snadal/BDIOntology/Source/> .

S:hasWrapper rdf:type rdf:Property ;
       rdfs:isDefinedBy <http://www.essi.upc.edu/~snadal/BDIOntology/Source/> ;
       rdfs:domain S:DataSource ;
       rdfs:range S:Wrapper .

S:hasAttribute rdf:type rdf:Property ;
       rdfs:isDefinedBy <http://www.essi.upc.edu/~snadal/BDIOntology/Source/> ;
       rdfs:domain S:Wrapper ;
       rdfs:range S:Attribute .
```

Code 7: Metadata model for $\mathcal{S}$ in Turtle notation

Example. Figure 4 shows the instantiation of $\mathcal{S}$ in SUPERSEDE. Red nodes depict the data sources that correspond to the three data sources introduced in Section 2.1. Then, orange and blue nodes depict the wrappers and attributes, respectively.

3.3. Mapping graph

As previously discussed, we encode LAV mappings in the ontology. Recall that mappings are composed by (a) subgraphs of $\mathcal{G}$, one per wrapper, and (b) the

[10] http://www.essi.upc.edu/~snadal/BDIOntology/Source

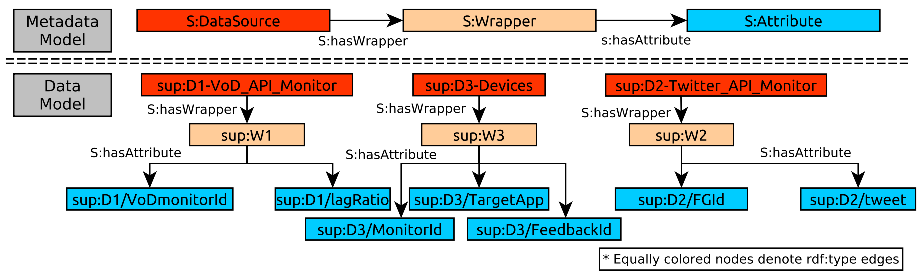

Figure 4: RDF dataset of the metadata model and data model of $\mathcal{S}$. For interpretation of the references to color in the text, the reader is referred to the web version of this article.

function F linking elements of type `S:Attribute` to elements of type `G:Feature`. We serialize such information in RDF in the Mapping graph $\mathcal{M}$. Subgraphs are represented using named graphs, which identify a subset of $\mathcal{G}$. Thus, each wrapper will have associated a named graph identifying which concepts and features it is providing information about. This will be represented using triples of the form $\langle w, \texttt{M:mapping}, G\rangle$, where w is an instance of `S:Wrapper` and G is a subgraph of $\mathcal{G}$. Regarding the function F, we represent it via the `owl:sameAs` property (i.e., triples of the form $\langle \mathsf{x}, \texttt{owl:sameAs}, y\rangle$, where x and y are respectively instances of `S:Attribute` and `G:Feature`.

Example. In Figure 5 we depict the complete instantiation of the BDI ontology for the SUPERSEDE running example. To ensure readability, internal classes are omitted and only the core ones are shown. Named graphs are depicted using colored boxes, respectively red for w_1, blue for w_2 and green for w_3.

The previous discussion sets the baseline to enable semi-automatic schema management in the data sources. Instantiating the metadata model, the data steward is capable of modeling the schema of the sources to be further linked to the wrappers and the data instances they provide. With such, in the rest of this paper we will introduce techniques to adapt the ontology to schema evolution aswell as query answering.

4. Handling evolution

In this section, we present how the BDI ontology accomodates the evolution of situational data. Specific studies concerning REST API evolution [14, 27] have concluded that most of such changes occur in the structure of incoming events, thus our goal is to semi-automatically adapt the BDI ontology to such evolution. To this end, in the following subsections we present an algorithm to aid the data steward to enrich the ontology upon new releases.

4.1. Releases

In Section 2, we discussed the role of the data steward as the unique maintainer of the BDI ontology in order to make data management tasks transparent to

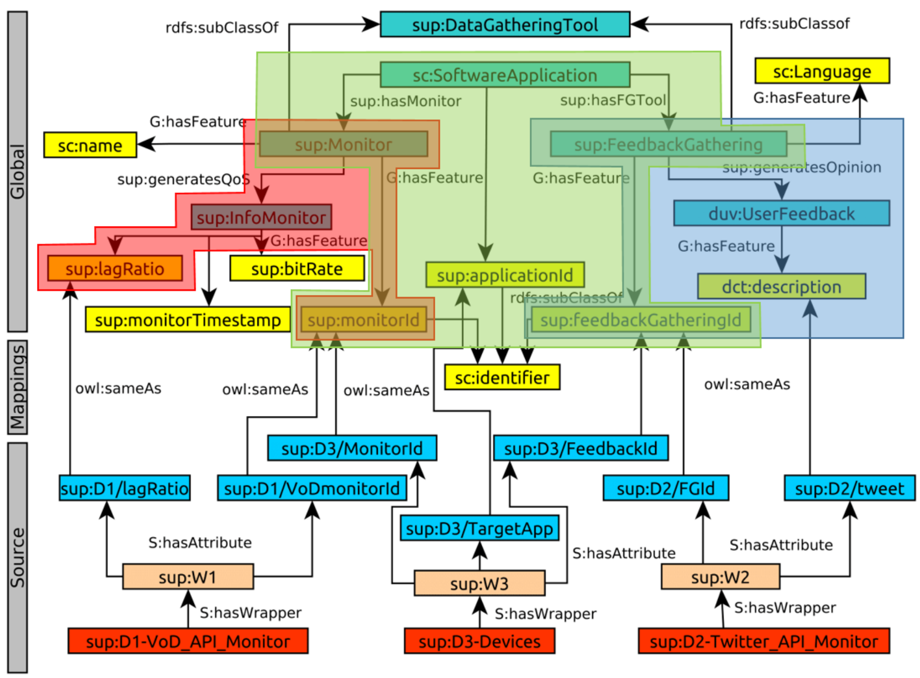

Figure 5: RDF dataset of the metadata model and data model of the complete ontology for the SUPERSEDE running example. For interpretation of the references to color in the text, the reader is referred to the web version of this article.

data analysts. Now, the goal is to shield the analysts queries, so that they do not crash upon new API version releases. In other words, we need to adapt $\mathcal{S}$ to schema evolution in the data sources, so that $\mathcal{G}$ is not affected. To this end, we introduce the notion of *release*, the construct indicating the creation of a new wrapper, and how its elements link to features in $\mathcal{G}$. Thus, we formally define a release R as a 3-tuple $R = \langle w, G, F\rangle$, where w is a wrapper, G is a subgraph of $\mathcal{G}$ denoting the elements in $\mathcal{G}$ that the wrapper contributes to, and $F = a \mapsto V(G)$ a function where $a \in w.\overline{a_{ID}} \cup w.\overline{a_{nID}}$ and $V(G)$ vertices of type `G:Feature` in $\mathcal{G}$. R must be created by the data steward upon new releases. Several approaches can aid this process. For instance, to define the graph G, the user can be presented with subgraphs of $\mathcal{G}$ that cover all features. However, this raises the question of which is the most appropiate subgraph that the user is interested in. Regarding the definition of F, probabilistic methods to align and match RDF ontologies, such as PARIS [25], can be used. Note that the definition of wrappers (i.e., how to query an API) is beyond the scope of this paper.

Example. Recall wrapper w_4 for data source D_1. Its associated release would be defined as $w_4(\mathsf{VoDmonitorId}, \mathsf{bufferingRatio})$, $G = \texttt{sup:lagRatio} \xleftarrow{\texttt{G:hasFeature}} \texttt{sup:InfoMonitor} \xrightarrow[\texttt{sup:hasMonitor}]{} \texttt{sup:Monitor} \xrightarrow[\texttt{G:hasFeature}]{} \texttt{sup:monitorId}$, and

$F = \{\textsf{VoDmonitorId} \mapsto \texttt{sup:monitorId}, \textsf{bufferingRatio} \mapsto \texttt{sup:lagRatio}\}$.

4.2. Release-based Ontology Evolution

As mentioned above, changes in the source elements need to be reflected in the ontology to avoid queries to crash. Furthermore, the ultimate goal is to provide such adaptation in an automated way. To this end, Algorithm 1 applies the necessary changes to adapt the BDI ontology $\mathcal{T}$ w.r.t. a new release R. It starts registering the data source, in case it is new (line 4), and the new wrapper to further link them (lines 7 and 8). Then, for each attribute in the wrapper $R.w$, we check their existence in the current Source graph and register it, in case it is not present. Given the way URIs for attributes are constructed (i.e., they have the prefix of their source), we can ensure that only attributes from the same source will be reused within subsequent versions. This helps to maintain a low growth rate for $\mathcal{T}.\mathcal{S}$, as well as avoiding potential semantic differences. Next, the named graph is registered to the Mapping graph, to conclude with the serialization of function F (in $R.F$). The complexity of this algorithm is linearly bounded by the size of the parameters of R.

Algorithm 1 Adapt to Release

Pre: $\mathcal{T}$ is the BDI ontology, R new release
Post: $\mathcal{T}$ is adapted w.r.t. R
1: **function** NewRelease($\mathcal{T}$, R)
2: $Source_{uri}$ = "S:DataSource/"+$source(R.w)$
3: **if** $Source_{uri} \notin$ SELECT $?ds$ FROM $\mathcal{T}$ WHERE $\langle ?ds$, "rdf:type", "S:DataSource"$\rangle$ **then**
4: $\mathcal{T}.\mathcal{S} \cup= \langle Source_{uri}$, "rdf:type", "S:DataSource"$\rangle$
5: **end if**
6: $Wrapper_{uri}$ = "S:Wrapper/"+$R.w$
7: $\mathcal{T}.\mathcal{S} \cup= \langle Wrapper_{uri}$, "rdf:type", "S:Wrapper"$\rangle$
8: $\mathcal{T}.\mathcal{S} \cup= \langle Source_{uri}$, "S:hasWrapper", $Wrapper_{uri}\rangle$
9: **for each** $a \in (R.w.\overline{a_{ID}} \cup R.w.\overline{a_{nID}})$ **do**
10: $Attribute_{uri} = Source_{uri}$+$a$
11: **if** $Attribute_{uri} \notin$ SELECT $?a$ FROM $\mathcal{T}$ WHERE $\langle ?a$, "rdf:type", "S:Attribute"$\rangle$ **then**
12: $\mathcal{T}.\mathcal{S} \cup= \langle Attribute_{uri}$, "rdf:type", "S:Attribute"$\rangle$
13: **end if**
14: $\mathcal{T}.\mathcal{S} \cup= \langle Wrapper_{uri}$, "S:hasAttribute", $Attribute_{uri}\rangle$
15: **end for**
16: $\mathcal{T}.\mathcal{M} \cup= \langle Wrapper_{uri}$, "M:mapping", $R.G\rangle$
17: **for each** $(a, f) \in R.F$ **do**
18: $a_{uri} = Source_{uri}$+a
19: f_{uri} = "G:Feature/"+f
20: $\mathcal{T}.\mathcal{M} \cup= \langle a_{uri}$, "owl:sameAs", $f_{uri}\rangle$
21: **end for**
22: **end function**

Example. In Figure 6, we depict the resulting ontology $\mathcal{T}$ after executing Algorithm 1 with the release for wrapper w_4.

5. Query answering

In this section, we present the algorithm for ontology-based query answering under LAV mappings with wrappers. To this end, we provide a query rewriting algorithm that, given a conjunctive query $Q_{\mathcal{G}}$ produces a union of conjunctive

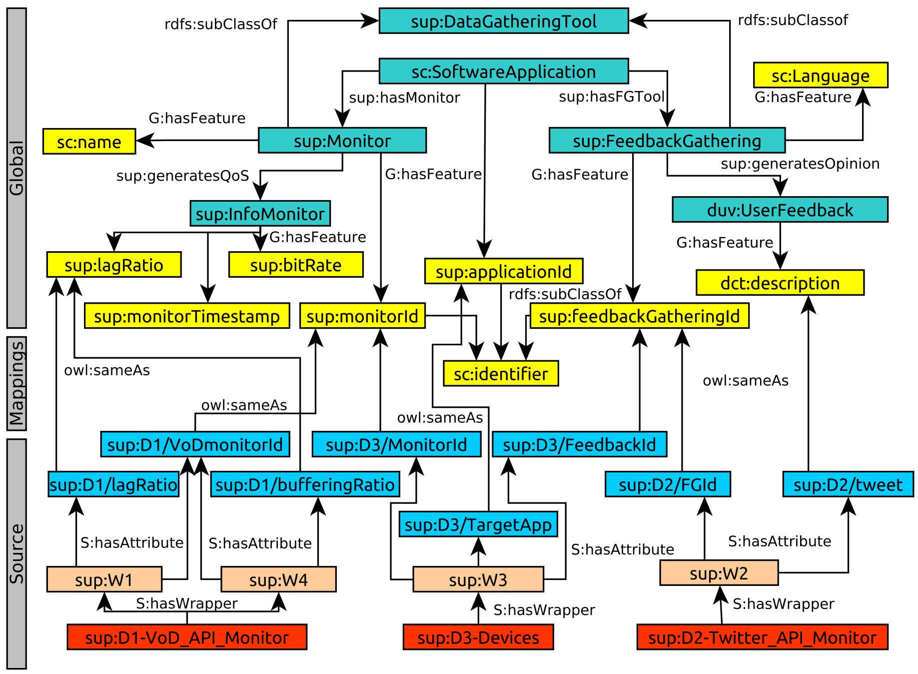

Figure 6: RDF dataset for the evolved ontology $\mathcal{T}$ for the SUPERSEDE running example

queries Q over the wrappers. Retaking the running example, and now using the vocabulary introduced in Section 3 as prefixes, the SPARQL representation of the query obtaining for each *applicationId* all its *lagRatio* instances would be that depicted in Code 8. Alternatively, recall the alternative representation for $Q_\mathcal{G}$ as $Q_\mathcal{G}.\pi = \{$`sup:applicationId`, `sup:lagRatio`$\}$ and the graph $Q_\mathcal{G}.\varphi$ depicted in Figure 7.

```
SELECT ?x ?y
FROM 𝒢
WHERE {
  VALUES (?x ?y) { (sup:applicationId sup:lagRatio) }
  sc:SoftwareApplication G:hasFeature sup:applicationId .
  sc:SoftwareApplication sup:hasMonitor sup:Monitor .
  sup:Monitor sup:generatesQoS sup:InfoMonitor .
  sup:InfoMonitor G:hasFeature sup:lagRatio
}
```

Code 8: Running example's SPARQL query

5.1. Well-formed queries

As previously mentioned, unambiguously resolving query answering under LAV mappings entails constraining the design of the elements in the ontology,

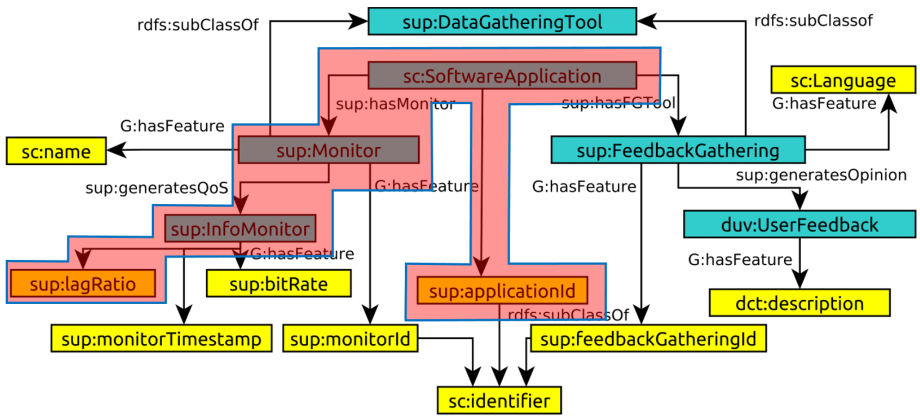

Figure 7: Graph pattern for the running example query

which also applies for the case of queries. Even though our approach makes transparent to the user how the concepts in $\mathcal{G}$ are to be joined in the wrappers, it is necessary that $Q.\pi$ retrieves only elements that exist in the sources (i.e., features) and can be populated with data. To this end, we introduce the notion of well-formed query.

Definition 5.1 (Well-formed query)**.** *A query $Q_\mathcal{G}$ is well formed iff $Q_\mathcal{G}.\varphi$ has a topological sorting (i.e., it is a DAG) and any projected element $p \in Q_\mathcal{G}.\pi$ refers to a terminal node $n \in Q_\mathcal{G}.\varphi$ which has a triple* $\langle n, \texttt{rdf:type}, \texttt{G:Feature}\rangle$ in $\mathcal{G}$.

The rationale behind such definition is to ensure that (a) the graph $Q_\mathcal{G}.\varphi$ can be safely traversed by joining different sources, and (b) all projected elements are features, which potentially have mappings to the sources. For instance, the SPARQL query depicted in Code 9, which retrieves pairs of *Monitor* and *FeedbackGathering* per *SoftwareApplication*, is not well-formed as it retrieves only concepts.

```
SELECT ?x, ?y, ?z
FROM 𝒢
WHERE {
  VALUES (?x ?y ?z) {
    (sup:SoftwareApplication sup:Monitor sup:FeedbackGathering)
  }
  sup:SoftwareApplication sup:hasMonitor sup:Monitor .
  sup:SoftwareApplication sup:hasFGTool sup:FeedbackGathering
}
```

Code 9: A non well-formed query

In our approach, IDs are considered the default feature. Hence, it is possible to automatically rewrite the query and make it well-formed by replacing projections of concepts for IDs, if available. Such process is depicted in Algorithm 2, which converts a query to a well-formed one if possible, otherwise it raises an error. Algorithm 2 firstly attempts to detect if the graph pattern $Q_\mathcal{G}.\varphi$ is acyclic, which will be true if and only if there exists a topological ordering. Next, it iterates over the projected elements in $Q_{\mathcal{G}}.\pi$ looking for those that are not of type `G:Feature` (line 6), in such case it explores all the features of the concept at hand looking for a candidate ID. Note the usage of the auxiliary method $x.\textsc{outgoingNeighborsOfType}(t, g)$, returning, for a node x, all outgoing neighbors of type t in the graph g (line 8). Code 10 depicts the previous non well-formed query now converted to its well-formed version after applying the algorithm.

Algorithm 2 Well-formed query

Pre: $\mathcal{T}$ is the BDI ontology, $Q_{\mathcal{G}} = \langle \pi, \varphi \rangle$ is a query over $\mathcal{G}$
Post: $Q_{\mathcal{G}}$ is well-formed, otherwise an error is raised

```
1: function WellFormedQuery(G, Q_G)
2:    if ∄TopologicalSort(Q_G.φ) then
3:       return error(Q_G.φ has at least one cycle)
4:    end if
5:    for each π ∈ Q_G.π do
6:       if typeOf(π) ≠ G:Feature then
7:          hasID = false
8:          for each o ∈ π.outgoingNeighborsOfType("G:Feature", T) do
9:             if ⟨o, "rdfs:subClassOf", "sc:identifier"⟩ ∈ T then
10:               hasID = true
11:               Q_G.π = (Q_G.π \ {π}) ∪ {o}
12:               Q_G.φ ∪= ⟨π, "G:hasFeature", o⟩
13:            end if
14:         end for
15:         if ¬hasID then
16:            return error(Q_G has at least one concept without any feature included in the query that is mapped to the sources)
17:         end if
18:      end if
19:   end for
20:   return S
21: end function
```

```
SELECT ?x ?y ?z
FROM G
WHERE {
  VALUES (?x ?y ?z) {
    (sup:applicationId sup:monitorId sup:feedbackGatheringId)
  }
  sup:SoftwareApplication sup:hasMonitor sup:Monitor .
  sup:SoftwareApplication sup:hasFGTool sup:FeedbackGathering .
  sup:SoftwareApplication G:hasFeature sup:applicationId .
  sup:Monitor G:hasFeature sup:monitorId .
  sup:FeedbackGathering G:hasFeature sup:feedbackGatheringId
}
```

Code 10: A well-formed query

5.2. Query rewriting

The core of the query answering method is the query rewriting algorithm that, given a well-formed query $Q_{\mathcal{G}}$ automatically resolves the LAV mappings and returns a union of conjunctive queries over the wrappers. Intuitively, the algorithm consists of three phases:

1. *Query expansion*, which deals with the analysis of the query w.r.t. the ontology. To this end, it takes as input a well-formed query $Q_{\mathcal{G}}$ in order to build its *expanded* version. An expanded query $Q'_{\mathcal{G}}$ contains the same elements as the original $Q_{\mathcal{G}}$, however it also includes IDs for concepts that have not been explicitely requested by the analyst. This is necessary to perform joins in the next phases. In this phase, we also identify which are the concepts in the query, as the next phases are concept-centric.

2. *Intra-concept generation*, which receives as input the expanded query and generates a list of *partial walks* per concept. Such partial walks indicate how to query the wrappers in order to obtain the requested features for the concept at hand. To achieve this, we utilize SPARQL queries that aid us to obtain the features per concept, as well as to resolve the LAV mappings.

3. *Inter-concept generation*, it receives the list of partial walks per concept and joins them to produce covering walks. As result, it returns the union of all the covering and minimal walks found. This is achieved by generating all combinations of partial conjunctive queries that can be joined and that cover the projected attributes in $Q_{\mathcal{G}}$.

Next, we present the algorithms corresponding to each of the phases and their details.

Phase #1 (query expansion). The expansion phase (see Algorithm 3) breaks down to the following steps:

(1) **Identify query-related concepts.** The list of query-related concepts consists of vertices of type `G:Concept` in the graph pattern (line 4). Traversing $Q_{\mathcal{G}}.\varphi$ we manage to store adjacent concepts in the query in the list *concepts* (line 5). For the sake of conciseness, algorithms assume linear traversals amongst concepts. Note that using tree-shaped concept traversals is possible, but entails overburdening the algorithms with graph manipulations instead of lists.

Example. In the running example (see Figure 7), the list *concepts* would be [`sc:SoftwareApplication`, `sup:Monitor`, `sup:InfoMonitor`].

(2) **Expand $Q_{\mathcal{G}}$ with IDs.** Given the list of query-related concepts, we identify their features of type ID by means of a SPARQL query and store it in the set IDs (line 10). For each element in the set IDs we finally expand the query with it (line 12).

Example. The expanded query $Q'_{\mathcal{G}}$ would include the feature `sup:monitorId` (i.e., the ID of concept `sup:Monitor`), which was not initially in $Q_{\mathcal{G}}$.

Algorithm 3 Query Expansion

Pre: $Q_{\mathcal{G}}$ is a well-formed query, $\mathcal{T}$ us the BDI ontology
Post: *concepts* is the list of query related concepts, $Q'_{\mathcal{G}}$ is the expanded version of $Q_{\mathcal{G}}$ with IDs

```
 1: function QueryExpansion(Q_G, G)
 2:   concepts = [ ]                                                     ①
 3:   for v ∈ TopologicalSort(Q_G.φ) do
 4:     if ⟨v, "rdf:type", "G:Concept"⟩ ∈ T then
 5:       concepts.add(v)
 6:     end if
 7:   end for
 8:   Q'_G = Q_G
 9:   for c ∈ concepts do                                                ②
10:     IDs = SELECT ?t FROM T WHERE
          {⟨c, "G:hasFeature", ?t⟩.⟨?t, "rdfs:subClassOf", "sc:identifier"⟩}
11:     for f_ID ∈ IDs do
12:       Q'_G.φ ∪= ⟨c, "G:hasFeature", f_ID⟩
13:     end for
14:   end for
15:   return ⟨concepts, Q'_G⟩
16: end function
```

Phase #2 (intra-concept generation). The intra-concept phase (see Algorithm 4) gets as input the list of concepts in the query, and the expanded query $Q'_{\mathcal{G}}$, and outputs the list of partial walks per concept ($partialWalks$ defined in line 2). A partial walk is a walk that is not yet traversing all the concepts required by the query. The process breaks down to the following steps:

(3) **Identify queried features.** Phase #2 starts iterating for each concept in the query. First, we define the auxiliary hashmap $PartialWalksPerWrapper$ (line 5), where its keys are wrappers and its values are walks. To populate this map, we obtain the requested features in $Q'_{\mathcal{G}}$ for the concept at hand, which is stored in the set $features$ that is obtained via a SPARQL query over the graph pattern $Q'_{\mathcal{G}}.\varphi$ (line 6).

Example. The set $features$ (result of the SPARQL query in line 6) would be {`sup:lagRatio`, `sup:monitorId`, `sup:applicationId`}.

(4) **Unfold LAV mappings.** Next, for each feature f in the set $features$, we look for wrappers whose LAV mapping contain it. This is achieved querying the named graphs in $\mathcal{T}$ (line 8). At this point, we have the information of which wrappers may provide the feature at hand.

Example. For the feature `sup:lagRatio` the identified set of wrappers would be {`sup:W1`}. Likewise, for the feature `sup:monitorId` the set {`sup:W1`, `sup:W3`} and for `sup:applicationId` the set {`sup:W3`}.

(5) **Find attributes in $\mathcal{S}$.** Now, for each wrapper w in the previously devised set of wrappers for feature f, with a SPARQL query (line 10) we find the attribute a in $\mathcal{S}$ that maps to the feature at hand (i.e., `owl:sameAs` relationship). This will be added to the hashmap $PartialWalksPerWrapper$, with key w and value $\widetilde{\Pi}_a(w)$.

Example. For feature `sup:lagRatio` and wrapper `sup:W1`, we would identify `sup:D1/lagRatio` as attribute in $\mathcal{S}$. Hence, we would add to the hashmap $PartialWalksPerWrapper$ an entry with key `sup:W1` and value $\widetilde{\Pi}_{\texttt{sup:D1/lagRatio}}(\texttt{sup:W1})$. The process would be likewise for the rest of features and wrappers.

(6) **Prune output.** Note that we might have considered walks that do not contain all the requested features for the current concept c (e.g., a wrapper w_5 where lagRatio has been dropped), hence, in order to avoid the complexity that combining wrappers within a concept would yield, we only keep those wrappers providing all the features queried for the current concept. To this end, we first use the MergeProjections operator, which merges the projection operators that have been separately added to the walk (e.g., from $\widetilde{\Pi}_{a_1}(w)\widetilde{\Pi}_{a_2}(w)$ to $\widetilde{\Pi}_{a_1,a_2}(w)$). With such wrapper projections, we follow the `owl:sameAs` relation from $\mathcal{S}$ to $\mathcal{G}$ to ensure that we are obtaining the same set of features as requested by the analyst (defined in line 6), if so we will add such partial walk to the output, ensuring *covering* and *minimality* for the concept at hand.

Example. The final output of phase #2 would be a list with the following elements:

- $\langle$ `sc:SoftwareApplication` $\rightarrow \{\widetilde{\Pi}_{\texttt{sup:D3/TargetApp}}(\texttt{sup:W3})\}\rangle$
- $\langle$ `sup:Monitor` $\rightarrow \{\widetilde{\Pi}_{\texttt{sup:D1/VoDmonitorId}}(\texttt{sup:W1}), \widetilde{\Pi}_{\texttt{sup:D3/MonitorId}}(\texttt{sup:W3})\}\rangle$
- $\langle$ `sup:InfoMonitor` $\rightarrow \{\widetilde{\Pi}_{\texttt{sup:D1/lagRatio}}(\texttt{sup:W1})\}\rangle$

Phase #3 (inter-concept generation). The final phase of the rewriting process (see Algorithm 5) consists of joining the partial walks per concept to obtain a set of walks joining all the concepts required in the query. This is a systematic process where the final list of walks is incrementally built.

(7) **Compute cartesian product.** Phase #3 iterates on $partialWalks$ using a window of two elements, *current* (line 2) and *next* (line 4), and maintain a set of currently joined partial walks (line 5). We start computing the cartesian product of the respective lists of partial walks (line 6), namely CP_{left} (corresponding to *current*) and CP_{right} (corresponding to *next*).

Example. In the first iteration, *current* and *next* would be respectively the maps $\langle$ `sc:SoftwareApplication` $\rightarrow \{\widetilde{\Pi}_{\texttt{sup:D3/TargetApp}}(\texttt{sup:W3})\}\rangle$ and $\langle$ `sup:Monitor` $\rightarrow \{\widetilde{\Pi}_{\texttt{sup:D1/VoDmonitorId}}(\texttt{sup:W1}), \widetilde{\Pi}_{\texttt{sup:D3/MonitorId}}(\texttt{sup:W3})\}\rangle$. Thus, the resulting cartesian product of the sets of partial walks would be the pair $\langle\widetilde{\Pi}_{\texttt{sup:D3/TargetApp}}(\texttt{sup:W3}), \widetilde{\Pi}_{\texttt{sup:D1/VoDmonitorId}}(\texttt{sup:W1})\rangle$ and the pair $\langle\widetilde{\Pi}_{\texttt{sup:D3/TargetApp}}(\texttt{sup:W3}), \widetilde{\Pi}_{\texttt{sup:D3/MonitorId}}(\texttt{sup:W3})\rangle$.

(8) **Merge walks.** Given the two partial walks from the cartesian product, the goal is now to merge them into a single one. To this end, we use the function MergeWalks (line 7) that given the two partial walks generates a merged one that projects the attributes from both inputs. At this moment

Algorithm 4 Intra-concept generation

Pre: *concepts* is the list of concepts in the query, $Q'_{\mathcal{G}}$ is an expanded query, $\mathcal{T}$ is the BDI ontology
Post: *partialWalks* is the map of sets of partial walks per concept
function IntraConceptGeneration($concepts, Q'_{\mathcal{G}}, \mathcal{T}$)
  $partialWalks = [\ ]$
  for $i = 0;\ i <$ LENGTH($concepts$); ++i **do**
    $c = concepts[i]$
    $PartialWalksPerWrapper$ = `HashMap<k,v>` 3
    $features$ = SELECT ?f FROM $Q'_{\mathcal{G}}.\varphi$ WHERE $\{\langle c,$ `"G:hasFeature"`$, ?f\rangle\}$
    for $f \in features$ **do**
      $wrappers$ = SELECT ?g FROM $\mathcal{T}$ WHERE
        { GRAPH ?$g\{\langle c,$ `"G:hasFeature"`$, f\rangle\}\}$ 4
      for $w \in wrappers$ **do**
        $attribute$ = SELECT ?a FROM $\mathcal{T}$ WHERE
          $\{\langle ?a,$ `"owl:sameAs"`$, f\rangle.\langle w,$ `"S:hasAttribute"`$, ?a\rangle\}$ 5
        $PartialWalksPerWrapper[w] \cup= \widetilde{\Pi}_{attribute}(w)$
      end for
    end for
    for $\langle wrapper, walk\rangle \in PartialWalksPerWrapper$ **do**
      $mergedWalk$ = MergeProjections($walk$)
      $featuresInWalk = \{\}$
      for $a \in$ Projections($mergedWalk$) **do**
        $featuresInWalk \cup=$ SELECT ?f FROM $\mathcal{T}$ WHERE
          $\{\langle a,$ `"owl:sameAs"`$, ?f\rangle\}$ 6
      end for
      if $featuresInWalk = features$ **then**
        $partialWalks$.ADD($\langle c, mergedWalk\rangle$)
      end if
    end for
  end for
  return $partialWalks$
end function

there are two possibilities, (a) there is a wrapper shared by both partial walks and then the join has been materialized by it, or (b) they do not share a wrapper, thus we need to explore ways to join them. In the former case, as discussed, no further join needs to be added to the merged walk, however the latter needs to be extended by an additional join ($\widetilde{\bowtie}$) between both inputs. Such discovery process is described in the following steps.

Example. Given $\langle\widetilde{\Pi}_{\texttt{sup:D3/TargetApp}}(\texttt{sup:W3}),\ \widetilde{\Pi}_{\texttt{D3/MonitorId}}(\texttt{sup:W3})\rangle$, the merged walk would be $\widetilde{\Pi}_{\texttt{sup:D3/TargetApp},\texttt{sup:D3/MonitorId}}(\texttt{sup:W3})$ where no extra joins should be added. Regarding the pair $\langle\widetilde{\Pi}_{\texttt{sup:D3/TargetApp}}(\texttt{sup:W3}),$ $\widetilde{\Pi}_{\texttt{sup:D1/VoDmonitorId}}(\texttt{sup:W1})\rangle$, after merging the walks the result would be $\widetilde{\Pi}_{\texttt{sup:D3/TargetApp}}(\texttt{sup:W3})\widetilde{\Pi}_{\texttt{sup:D1/VoDmonitorId}}(\texttt{sup:W1})$, thus it is necessary to discover how to join `sup:W1` and `sup:W3`.

9 **Discover join wrappers.** For each pair of concepts related by an edge in $Q'_{\mathcal{G}}$ (*current* and *next*), we aim at retrieving the list of wrappers providing the required features (i.e., identified as partial walks in the previous step). Since $\mathcal{G}$ is a directed graph, we first need to identify, for each edge, the concept playing the role of *current* and *next* (e.g., if `sc:SoftwareApplication` and `sup:Monitor` play the role of *current* and *next*, respectively, then the join must be computed using the ID of *next*).

This is computed in two SPARQL queries (lines 9 and 10). Note that only one direction will be available since our graph query ($Q'_{\mathcal{G}}$) does not contain cycles.

Example. Given that *current.c* and *next.c* are respectively the concepts `sc:SoftwareApplication` and `sup:Monitor`, as the edge is directed from the former to the latter, only *wrappersFromLtoR* would contain any data, precisely the set of wrappers {`sup:W1`}. This entails that we need to look for the attribute of type ID for concept `sup:Monitor` that is provided by `sup:W1`.

10 **Discover join attribute.** Focusing on the case where *next* must provide the ID (lines 12-17), we start issuing a SPARQL query that tells us such ID (line 12). Next, the operation findWrapperWithID (line 13) identifies which wrapper is providing such ID for *next*, and subsequently we obtain the physical attribute (line 14). Then, we iterate on all wrappers that contribute to the relation between both concepts, and for each wrapper we identify the ID attribute for $left$ (line 16). With such, we can generate a new walk by joining each potential pair resulting from the list of IDs for *current* and the one identified for *next* (line 17). As we previously discussed, this process depends on the direction of the edge, therefore line 20 entails that the same process should be executed if the edge goes from *next* to *current*.

Example. Given the partial walks from the previous example, the output of phase #3 would consist of the following set of walks:

- $\widetilde{\Pi}_{\texttt{sup:D1/lagRatio},\texttt{sup:D1/VoDmonitorId},\texttt{sup:D3/TargetApp}}$ $(\texttt{sup:W1} \underset{\texttt{sup:D1/VoDmonitorId}=\texttt{sup:D3/MonitorId}}{\widetilde{\bowtie}} \texttt{sup:W3})$
- $\widetilde{\Pi}_{\texttt{sup:D1/lagRatio},\texttt{sup:D3/MonitorId},\texttt{sup:D3/TargetApp}}$ $(\texttt{sup:W1} \underset{\texttt{sup:D1/VoDmonitorId}=\texttt{sup:D3/MonitorId}}{\widetilde{\bowtie}} \texttt{sup:W3})$

Note that, even though the analyst requested only the first and third attributes our approach has generated further combinations when considering IDs (in Step 2). Those can be easily projected out at the final step, when generating the union of conjunctive queries.

5.3. Computational complexity

The query rewriting algorithm is divided into three blocks, hence we will present the study of the computational complexity for each of them. We will study the complexity in terms of the number of walks generated in the worst case. Such worst case occurs when each concept features is provided by a different wrapper (which forces us to generate more joins) and for each concept different sources provide wrappers for it (which generates unions of alternative walks), which forces us to generate a larger number of joins.

Algorithm 5 Inter-concept generation

```
Pre: partialWalks is the list of partial walks per concept, S is the source graph and M the LAV
    mappings
Post: walks is the final list of walks
 1: function InterConceptGeneration(partialWalks, S, M)
 2:   current = partialWalks[0]
 3:   for i = 1; i < length(partialWalks); ++i do
 4:     next = partialWalks[i]
 5:     joined = {}
 6:     for ⟨CP_left, CP_right⟩ ∈ current.lw × next.lw do                              } 7
 7:       mergedWalk = mergeWalks(CP_left, CP_right)                                   } 8
 8:       if wrappers(CP_left) ∩ wrappers(CP_right) = ∅ then
 9:         wrappersFromLtoR = SELECT ?g FROM T WHERE                                  }
              { GRAPH ?g {⟨current.c, ?x, next.c⟩}}                                    } 9
10:         wrappersFromRtoL = SELECT ?g FROM T WHERE                                  }
              { GRAPH ?g {⟨next.c, ?x, current.c⟩}}                                    }
11:         if wrappersFromLtoR ≠ ∅ then
12:           f_ID = SELECT ?t FROM T WHERE                                            }
                {⟨next.c, G:hasFeature, ?t⟩.⟨?t, rdfs:subClassOf, sc:identifier⟩}      }
13:           wrapperWithID_right = findWrapperWithID(CP_right)                        }
14:           att_right = SELECT ?a FROM T WHERE                                       }
                {⟨?a, owl:sameAs, f_ID⟩.⟨wrapperWithID_right, S:hasAttribute, ?a⟩}     }
                                                                                       } 10
15:           for w ∈ wrappersFromLtoR do                                              }
16:             att_left = SELECT ?a FROM T WHERE                                      }
                  {⟨?a, owl:sameAs, f_ID⟩.⟨w, S:hasAttribute, ?a⟩}                     }
17:             mergedWalk ∪= w ⋈~_{att_left=att_right} wrapperWithID_right            }
18:           end for
19:         else if wrappersFromRtoL ≠ ∅ then
20:           Repeat the process from lines 12-17 inverting left and right.
21:         end if
22:       end if
23:       joined.add(mergedWalk)
24:     end for
25:     current = ⟨next.c, joined⟩
26:   end for
27:   return current
28: end function
```

- Phase #1: this phase expands the query with IDs not explicitly queried and therefore it is linear in the number of concepts in the query.
- Phase #2: this phase is linear in the number wrappers providing all the required features of a given concept of the query. This complexity results from the fact that either a wrapper provides all the features of a concept or it is not considered. Thus, no combinations between wrappers are performed to obtain the features or a given concept. Thus, the output of such phase is an array, where each of its buckets is the size of the number of wrappers per concept ($[(\#W)_{C_1}, (\#W)_{C_2}, \ldots, (\#W)_{C_n}]$).
- Phase #3: this phase yields an exponential complexity as it generates joins of partial walks. Note that a cartesian product is performed for each partial walk of a given concept c in the query. Hence, in the worst case (i.e., all partial walks can be joined), we are generating all combinations of wrappers in order to join them (i.e., $(\#W)_{C_1} \times (\#W)_{C_2} \times \ldots \times (\#W)_{C_n}$).

With the previous discussion, we conclude that in the worst case we can upper bound the theoretical complexity to $\mathcal{O}(W^C)$, assuming each concept has W wrappers generating partial walks (see phase 2), and the query navigates over C concepts. Indeed, such complexity depends on the number of mappings that refer to the query subgraph. To verify the theoretical complexity we have performed a controlled experiment. We have constructed an artificial query navigating through 5 concepts and we have progressively increased the number of wrappers per concept from 1 to 25. Then, we measured the time needed to run the algorithms. This is depicted in Figure 8, the theoretical prediction (thin line) closely aligns with the observed performance (thick line). Despite the exponential behavior of query answering, we advocate that realistic Big Data scenarios (e.g., the SUPERSEDE running example) where data are commonly ingested in the form of events, such disjointness in wrappers amongst concepts is not common. In that case, there are few combinations to walk through edges in $\mathcal{G}$, and thus query answering remains tractable in practice.

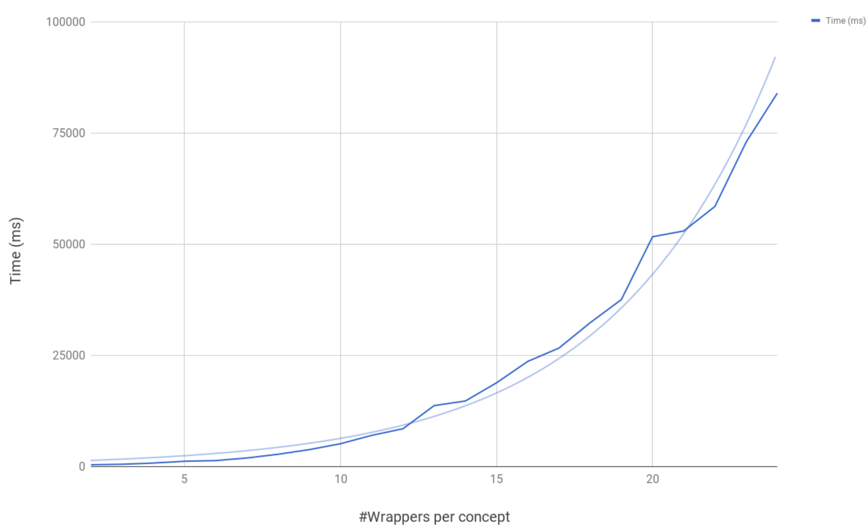

Figure 8: Evolution of query answering time in the worst case scenario where wrappers are disjoint (i.e., there is no evolution). The query is a query with 5 concepts. The x-axis shows the number of (disjoint) wrappers per concept.

6. Evaluation

In this section, we present the evaluation results of our approach. We first discuss its implementation, and then provide three kinds of evaluations: a functional evaluation on evolution management, the industrial applicability of our approach and a study on the evolution of the ontology in a real-world API.

6.1. Implementation

Prior to discuss the evaluation of our approach we present its implementation, which is part of a system named Metadata Management System (shortly MDM). Figure 9 depicts a functional overview of the querying process in the system. Data analysts are presented with a graph-based representation of $\mathcal{G}$ in a user interface where they can graphically pose OMQs. Such graphical representation is automatically converted to its equivalent SPARQL query, and if its well-defined to its algebraic expression $Q_{\mathcal{G}}$. Next, this is the input to our three-phase algorithm for query answering, which will yield a list of walks (i.e., relational algebra expressions over the wrappers).

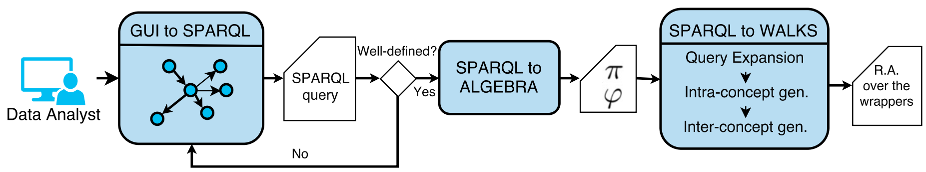

Figure 9: Architectural overview of the query answering process

MDM is implemented using a service-oriented architecture. In the frontend, it provides the web-based component to assist the management of the Big Data evolution lifecycle. This component is implemented in JavaScript and resides in a Node.JS web server, Figure 10 depicts an screenshot of the interface to query $\mathcal{G}$. The backend is implemented as a set of REST APIs defined with Jersey for Java. The backend makes heavy use of Jena to deal with RDF graphs, as well as its persistence engine Jena TDB.

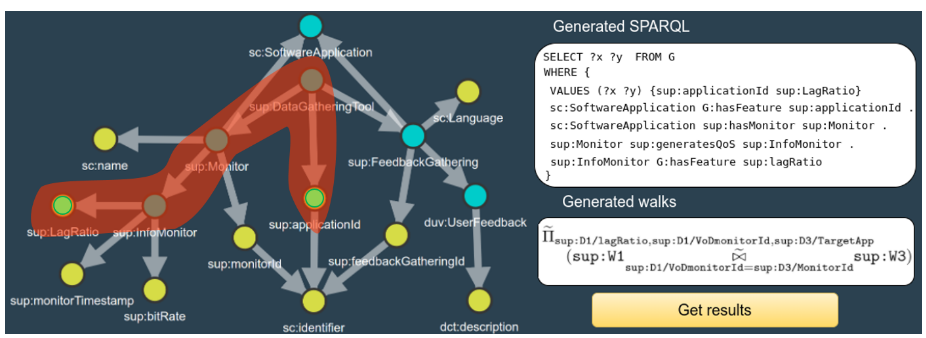

Figure 10: Posing an OMQ through the interface and the generated output

6.2. Functional evaluation

In order to evaluate the functionalities provided by the BDI ontology, we take the most recent study on structural evolution patterns in REST API [27]. Such work distinguishes changes at 3 different levels, those in (a) API-level, (b) method-level and (c) parameter-level. Our goal is to demostrate that our approach can semi-automatically accommodate such changes. To this end, it is necessary to make a distinction between those changes occurring in the data requests and those in the response. The former are handled by the wrapper's underlying query engine, which also needs to deal with other aspects such as authentication or HTTP query parametrization. The latter will be handled by the proposed ontology.

API-level changes. Those changes concern the whole of an API. They can be observed either because a new data source is incorporated (e.g., a new social network in the SUPERSEDE use case) or because all methods from a provider have been updated. Table 3 depicts the API-level change breakdown and the component responsible to handle it.

API-level Change	**Wrapper**	**BDI Ont.**
Add authentication model	✓	
Change resource URL	✓	
Change authentication model	✓	
Change rate limit	✓	
Delete response format		✓
Add response format		✓
Change response format		✓

Table 3: API-level changes dealt by wrappers or BDI ontology

Adding or changing a response format at API level consists of, for each wrapper querying it, registering a new release with this format. Regarding the deletion of a response format, it does not require actions, due to the fact that no further data on such format will arrive. However, in order to preserve historic backwards compatibility, no elements should be removed from $\mathcal{T}$.

Method-level changes. Those changes concern modifications on the current version of an operation. They occur either because a new functionality is released or because existing functionalities are modified. Table 4 summarizes the method-level change breakdown and the component responsible to handle it.

Method-level Change	**Wrapper**	**BDI Ont.**
Add error code	✓	
Change rate limit	✓	
Change authentication model	✓	
Change domain URL	✓	
Add method	✓	✓
Delete method	✓	✓
Change method name	✓	✓
Change response format		✓

Table 4: Method-level changes dealt by wrappers or BDI ontology

Those changes have more overlapping with the wrappers due to the fact that new methods require changes in both request and response. In the context of the BDI ontology, each method is an instance of `S:DataSource` and thus, adding a new one consists of declaring a new release and running Algorithm 1. Renaming a method requires renaming the data source instance. As before, a removal does not entail any action with the aim of preserving backwards historic compatibility.

Parameter-level changes. Such changes are those concerning schema evolution and are the most common on new API releases. Table 5 depicts such changes and the component in charge of handling it.

Parameter-level Change	**Wrapper**	**BDI Ont.**
Change rate limit	✓	
Change require type	✓	
Add parameter	✓	✓
Delete parameter	✓	✓
Rename response parameter		✓
Change format or type		✓

Table 5: Parameter-level changes dealt by wrappers or BDI ontology

Similarly to the previous level, some parameter-level changes are managed by both wrappers and the ontology. This is caused by the ambiguity of the change statements, and hence we might consider both URL query parameters and response parameters (i.e., attributes). Changing format of a parameter has a different meaning as before, and here entails a change of data type or structure. Any of the parameter-level changes identified can be automatically handled by the same process of creating a new release for the source at hand.

6.3. Industrial applicability

After functionally validating that the BDI ontology and wrappers can handle all types of API evolution, next we aim to study how these changes occur in real-world APIs. With this purpose, we study the results from [14] which presents 16 change patterns that frequently occur in the evolution of 5 widely used APIs (namely *Google Calendar*, *Google Gadgets*, *Amazon MWS*, *Twitter API* and *Sina Weibo*). With such information, we can show the number of changes per API that could be accommodated by the ontology. We summarize the results in Table 6. As before, we distinguish between changes concerning (a) the wrappers, (b) the ontology and (c) both wrappers and ontology. This enables us to measure the percentage of changes per API that can be partially accommodated by the ontology (changes also concerning the wrappers) and those fully accommodated (changes only concerning the ontology). Our results show that for all studied APIs, the BDI ontology could, on average, partially accommodate 48.84% of changes and fully accommodate 22.77% of changes. In other words, our semi-automatic approach allows to solve on average 71.62% of changes.

API Owner	#Changes Wrapper	#Changes Ontology	#Changes Wrapper&Ontology	Partially Accommodates	Fully Accommodates
Google Calendar	0	24	23	48.94%	51.06%
Google Gadgets	2	6	30	78.95%	15.79%
Amazon MWS	22	36	14	19.44%	50%
Twitter API	27	0	25	48.08%	0%
Sina Weibo	35	3	56	59.57%	3.19%

Table 6: Number of changes per API and percentage of partially and fully accommodated changes by $\mathcal{T}$

6.4. Ontology evolution

Now, we are concerned with performance aspects of using the ontology. Particularly, we will study its temporal growth w.r.t. the releases of a real-world API, namely Wordpress REST API[11]. This analysis is of special interest, considering that the size of the ontology may have a direct impact on the cost of querying and maintaining it. As a measure of growth, we count the number of triples in $\mathcal{S}$ after each new release, as it is the most prone to change. Given the high complexity of such APIs, we focus on a specific method and study its structural changes, namely the *GET Posts* API. By studying the changelog, we start from the currently deprecated version 1 evolving it to the next major version release 2. We further introduce 13 minor releases of version 2. (the details of the analysis can be found in [19]). We assume that a new wrapper providing all attributes is defined for each release.

The barcharts in Figure 11 depict the number of triples added to $\mathcal{S}$ per version release. As version 1 is the first occurrence of such endpoint, all elements must be added and thus carries a big overhead. Version 2 is a major release where few elements can be reused. Later, minor releases do not have many schema changes, with few attribute additions, deletions or renames. Thus, the largest batch of triples per minor release are edges of type `S:hasAttribute`. Each new version needs to identify which attributes it provides even though no change has been applied to it w.r.t. previous versions.

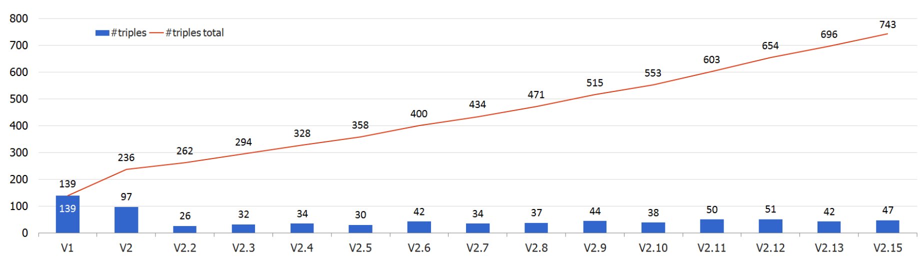

Figure 11: Growth in number of triples for $\mathcal{S}$ per release in Wordpress API

With such analysis we conclude that major version changes entail a steep

[11] https://wordpress.org/plugins/rest-api

growth, however that is infrequent in the studied API. On the other hand, minor versions occur frequently but the growth in terms of triples has a steady linear growth. The red line depicts the cumulative number of triples after each release. For a practically stable amount of minor release versions, we obtain a linear, stable growth in $\mathcal{S}$. Notice also that $\mathcal{G}$ does not grow. Altogether guarantees that querying $\mathcal{T}$ in query answering will not impose a big overhead, ensuring a good performance of our approach across time. Nonetheless, other optimization techniques (e.g., caching) can be used to further reduce the query cost.

7. Related work

In previous sections, we have cited relevant works on RESTful API evolution [27, 14]. They provide a catalog of changes, however they do not provide any approach to systematically deal with them. Other similar works, such as [28], empirically study API evolution aiming to detect its healthiness. If we look for approaches that automatically deal with such evolution, we must shift the focus to the area of database schemas, which are mostly focused on relational databases [24, 17]. They apply view cloning to accommodate changes while preserving old views. Such techniques rely on the capability of vetoing certain changes that might affect the overall integrity of the system. This is however an unrealistic approach to adopt in our setting, as schema changes are done by third party data providers.

Attention has also been paid to change management in the context of description logics (DLs). The definition of a DL that provides expresiveness to represent temporal changes in the ontology has been an interesting topic of study in the past years [16]. Relevant examples include [3], that defines the temporal DL *TQL*, providing temporal aspects at the conceptual model level, or [10] that delves on how to provide such temporal aspects for specific attributes in a conceptual model. It is known, however, that providing such temporal aspects to DLs entails a poor computational behaviour for CQ answering [16], for instance the previous examples are respectively coNP-*hard* and undecidable. Recent efforts are being put to overcome such issues and to provide tractable DLs and methods for rewritability of OMQs. For instance, [2] provides a temporal DL where the cost of first-order rewritability is polynomial, however that is only applicable for a restricted fragment of *DL-Lite*, and besides the notion of temporal attribute, which is key for management of schema evolution does not exist. Generally speaking, most of this approaches lack key characteristics for the management of schema evolution [21].

Regarding LAV schema mappings in data integration, few approaches strictly follow its definition. This is mostly due to the inherent complexity of query answering in LAV, which is reduced to the problem of answering queries using views [13]. Probably the most prominent data integration system that follows the LAV approach is Information Manifold [11]. To overcome the complexity posed by LAV query answering, combined approaches of GAV and LAV have been proposed, which are commonly referred as *both-as-view* (BAV) [18] or *global-and-local-as-view* (GLAV) [6]. Oppositely, we are capable of adopting a purely LAV approach by restricting the kind of allowed queries as well as how the mediated schema (i.e., ontology) has to be constructed.

Novelty with respect to the state of the art. Going beyond the related literature on management of schema evolution, our DOLAP'17 paper [20] proposed an RDF vocabulary-based approach to tackle such kind of evolution. Precisely, we focused on Big Data ecosystems that ingest data from REST APIs in JSON format. This paper extends our prior work, where, in the line of the mediator/wrapper architecture, we delegate the complexity of querying the sources to the wrappers. With such, we achieve the possibility to define LAV mappings, which are required in our setting. More importantly, we provide a tractable query answering algorithm that does not require reasoning to resolve LAV mappings.

8. Conclusions and Future Work

Our research aims at providing self-adapting capabilities in the presence of evolution in Big Data ecosystems. In this paper, we have presented the building blocks to handle schema evolution using a vocabulary-based approach to OBDA. Thus, unlike current OBDA approaches, we restrict the language from generic knowledge representation ontology languages (such as DL-Lite) to ontologies based on RDF vocabularies. We also restrict reasoning to the RDFS entailment regime. These decisions are made to enable LAV mappings instead of GAV. The proposed Big Data integration ontology aims to provide data analysts with an RDF-based conceptual model of the domain of interest, with the limitations that features cannot be reused among concepts. Data sources are accessed via wrappers, which must expose a relational schema in order to depict its RDF-based representation in the ontology and define LAV mappings, by means of named graphs and links from attributes to features. We have defined a query answering algorithm that leverages the proposed ontology and translates a restricted subset of SPARQL queries (see Section 2.2) over the ontology to queries over the sources (i.e., relational expressions on top of the wrappers). Also, we have presented an algorithm to aid data stewards to systematically accommodate announced changes in the form of releases. Our evaluation results show that a great number of changes performed in real-world APIs could be semi-automatically handled by the wrappers and the ontology. We additionally have shown the feasability of our query answering algorithm. There are many interesting future directions. A prominent one is to extend the ontology with richer constructs to semi-automatically adapt to unanticipated schema changes.

Acknowledgements

We thank the reviewers of both this paper and of its earlier version for their constructive comments that have significantly improved the quality of the paper. This work was partly supported by the H2020 SUPERSEDE project, funded by the EU Information and Communication Technologies Programme under grant agreement no 644018, and the GENESIS project, funded by the Spanish Ministerio de Ciencia e Innovación under project TIN2016-79269-R.

9. References

[1] A. Abelló, J. Darmont, L. Etcheverry, M. Golfarelli, J. Mazón, F. Naumann, T. B. Pedersen, S. Rizzi, J. Trujillo, P. Vassiliadis, and G. Vossen. Fusion Cubes: Towards Self-Service Business Intelligence. *IJDWM*, 9(2):66–88, 2013.

[2] A. Artale, R. Kontchakov, A. Kovtunova, V. Ryzhikov, F. Wolter, and M. Zakharyaschev. First-order rewritability of temporal ontology-mediated queries. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 2706–2712, 2015.

[3] A. Artale, R. Kontchakov, F. Wolter, and M. Zakharyaschev. Temporal description logic for ontology-based data access. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 711–717, 2013.

[4] P. Downey. XML Schema Patterns for Common Data Structures. *W3.org*, 2005.

[5] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[6] M. Friedman, A. Y. Levy, and T. D. Millstein. Navigational plans for data integration. In *Proceedings of the IJCAI-99 Workshop on Intelligent Information Integration, Held on July 31, 1999 in conjunction with the Sixteenth International Joint Conference on Artificial Intelligence City Conference Center, Stockholm, Sweden*, 1999.

[7] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book (2. ed.).* Pearson Education, 2009.

[8] I. Horrocks, M. Giese, E. Kharlamov, and A. Waaler. Using semantic technology to tame the data variety challenge. *IEEE Internet Computing*, 20(6):62–66, 2016.

[9] P. Jovanovic, O. Romero, and A. Abelló. A Unified View of Data-Intensive Flows in Business Intelligence Systems: A Survey. *T. Large-Scale Data- and Knowledge-Centered Systems*, 29:66–107, 2016.

[10] C. M. Keet and E. A. N. Ongoma. Temporal attributes: Status and subsumption. In *11th Asia-Pacific Conference on Conceptual Modelling, APCCM 2015, Sydney, Australia, January 2015*, pages 61–70, 2015.

[11] T. Kirk, A. Y. Levy, Y. Sagiv, D. Srivastava, et al. The information manifold. In *Proceedings of the AAAI 1995 Spring Symp. on Information Gathering from Heterogeneous, Distributed Enviroments*, volume 7, pages 85–91, 1995.

[12] M. Lenzerini. Data Integration: A Theoretical Perspective. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 233–246, 2002.

[13] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California, USA*, pages 95–104, 1995.

[14] J. Li, Y. Xiong, X. Liu, and L. Zhang. How Does Web Service API Evolution Affect Clients? In *2013 IEEE 20th International Conference on Web Services, Santa Clara, CA, USA, June 28 - July 3, 2013*, pages 300–307, 2013.

[15] A. Löser, F. Hueske, and V. Markl. Situational business intelligence. In *Business Intelligence for the Real-Time Enterprise - Second International Workshop, BIRTE 2008, Auckland, New Zealand, August 24, 2008, Revised Selected Papers*, pages 1–11, 2008.

[16] C. Lutz, F. Wolter, and M. Zakharyaschev. Temporal description logics: A survey. In *15th International Symposium on Temporal Representation and Reasoning, TIME 2008, Université du Québec à Montréal, Canada, 16-18 June 2008*, pages 3–14, 2008.

[17] P. Manousis, P. Vassiliadis, and G. Papastefanatos. Impact Analysis and Policy-Conforming Rewriting of Evolving Data-Intensive Ecosystems. *J. Data Semantics*, 4(4):231–267, 2015.

[18] P. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 227–238, 2003.

[19] S. Nadal, O. Romero, A. Abelló, P. Vassiliadis, and S. Vansummeren. Wordpress Evolution Analysis `www.essi.upc.edu/~snadal/wordpress_evol.txt`, 2016.

[20] S. Nadal, O. Romero, A. Abelló, P. Vassiliadis, and S. Vansummeren. An integration-oriented ontology to govern evolution in big data ecosystems. In *Proceedings of the Workshops of the EDBT/ICDT 2017 Joint Conference (EDBT/ICDT 2017), Venice, Italy, March 21-24, 2017.*, 2017.

[21] N. F. Noy and M. C. A. Klein. Ontology evolution: Not the same as schema evolution. *Knowl. Inf. Syst.*, 6(4):428–440, 2004.

[22] C. Pautasso, O. Zimmermann, and F. Leymann. Restful Web Services vs. "Big" Web Services: Making The Right Architectural Decision. In *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*, pages 805–814, 2008.

[23] A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Linking data to ontologies. *J. Data Semantics*, 10:133–173, 2008.

[24] I. Skoulis, P. Vassiliadis, and A. V. Zarras. Growing Up with Stability: How Open-source Relational Databases Evolve. *Inf. Syst.*, 53:363–385, 2015.

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

[26] H. J. ter Horst. Extending the RDFS entailment lemma. In *The Semantic Web - ISWC 2004: Third International Semantic Web Conference,Hiroshima, Japan, November 7-11, 2004. Proceedings*, pages 77–91, 2004.

[27] S. Wang, I. Keivanloo, and Y. Zou. How Do Developers React to RESTful API Evolution? In *Service-Oriented Computing - 12th International Conference, ICSOC 2014, Paris, France, November 3-6, 2014. Proceedings*, pages 245–259, 2014.

[28] A. V. Zarras, P. Vassiliadis, and I. Dinos. Keep Calm and Wait for the Spike! Insights on the Evolution of Amazon Services. In *Advanced Information Systems Engineering - 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13-17, 2016. Proceedings*, pages 444–458, 2016.