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

Sergi Nadal
Universitat Politècnica de Catalunya, BarcelonaTech
snadal@essi.upc.edu

Oscar Romero
Universitat Politècnica de Catalunya, BarcelonaTech
oromero@essi.upc.edu

Alberto Abelló
Universitat Politècnica de Catalunya, BarcelonaTech
aabello@essi.upc.edu

Panos Vassiliadis
University of Ioannina
pvassil@cs.uoi.gr

Stijn Vansummeren
Université Libre de Bruxelles
svsummer@ulb.ac.be

ABSTRACT

Big Data architectures allow to flexibly store and process heterogeneous data, from multiple sources, in its original format. The structure of those data, commonly supplied by means of REST APIs, is continuously evolving, forcing data analysts using it need to adapt their analytical processes after each release. This gets more challenging when aiming to perform an integrated or historical analysis of multiple sources. To cope with such complexity, in this paper we present the Big Data Integration ontology, the core construct for a data governance protocol that systematically annotates and integrates data from multiple sources in its original format. To cope with syntactic evolution in the sources, we present an algorithm that semi-automatically adapts the ontology upon new releases. A functional evaluation on real-world APIs is performed in order to validate our approach.

CCS Concepts

•Information systems → Mediators and data integration; Stream management; •Software and its engineering → Software evolution;

Keywords

Modeling, Semi-Structured Data, Evolution, Stream Data, Semantic Web

1. INTRODUCTION

Big Data ecosystems enable organizations to evolve their decision making processes from classic stationary data analysis [2] (e.g., transactional) to include situational data [12] (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 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 [17], 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 drawback for the analysts on the other side. As opposite to other protocols offering machine-readable contracts for the structure of the provided data (e.g., SOAP), web services using REST do not publish such information. Hence, *analysts need to go over the tedious task of carefully studying the documentation and adapting their tools 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 tools to such changes.*

Providing an integrated view over such evolving and heterogeneous set of data sources is a challenging problem which current Big Data technologies fail to address [1]. Take for instance the λ-architecture [13], the most widespread framework for Big Data systems. By dividing the processing pipeline into the Speed and Batch layers, it enables to perform both real-time and historical data analysis. Even though it enables to easily ingest, store and process situational data, it lacks a component providing an integrated global view or schema. To this end, in this paper we present an approach that, in the context of a λ-architecture, enables data analysts to (a) integrate situational data coming from external providers, as well as (b) smoothly facilitate the co-evolution of data and analytical processes preserving backward compatibility. Following the many "V's" Big Data definition, the former concerns *Variety* while the latter concerns *Variability.*

Case Study.

As an exemplar use case take the H2020 SUPERSEDE project[3], which we will use as our reference example throughout the paper. It aims to support decision-making in the evolution and adaptation of software services and applications by exploiting monitored end-user feedback and runtime data, with the overall goal of improving end-users' quality of experience (QoE). For the sake of this case study, we

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

narrow the scope to monitored data related to video on demand (VoD) and textual feedback from social networks (e.g., Twitter) in JSON format, as exemplified in Code 1.

```
{
  "VoDmonitorId": 12,
  "timestamp": 1475010424,
  "bitrate": 6,
  "rebufferingTimes": 2,
  "lagRatio": 75
}

{
  "feedbackGatheringId": 25,
  "lang": "en",
  "tweet": "I continuosly see the loading symbol",
  "created_at": 1475010415,
  "hashtags": ["#videoPlayer"],
  "user":  {
    "id_str": 12,
    "screen_name": "John"
  }
}
```

Code 1: Sample events for VoD and Twitter feedback monitors respectively

Assume that data analysts use a complex QoE metric from the integrated analysis over VoD and Twitter monitors. VoD monitors use `lagRatio` as a measure for the quality of service, measuring the percentage of time a user is waiting for a video. Second, we can quantify the satisfaction of a user by extracting the sentiment from the tweet s/he has issued. Hence, we define $QoE = (1 - lagRatio/100) \cdot sentiment(text)$ yielding a bounded value between 0 and 1. For instance, the tweet in Code 1 has a positive sentiment of 0.6[4], however with the integrated analysis and a lag ratio of 75% we would obtain a QoE score of 0.15, indicating a quality decrease.

Coming to the essence of our contribution, which involves the management of evolution, assume now that Twitter Search API[5] upgrades from version 1.0 to 1.1. Even if Twitter issues an announcement beforehand, and even if our feedback monitor is adapted to the new version (e.g., renaming the `tweet` attribute to `text`), still, all data analysts performing integrated QoE analysis will see their processes crash as they are not syntactically valid anymore, with the hassle of fixing them to conform the new schema.

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.

The problem is not straightforwardly addressable, despite the valiant efforts of the research community. Previous work on schema evolution has focused on software obtaining data from relational views [11, 21]. Such approaches rely on the capacity to veto changes affecting consumer applications. Those techniques are not valid in our setting given the lack of explicit schema information, as well as the impossibility to prevent changes from third party data providers.

So, to address the problem, we introduce the Big Data Integration ontology that (a) enables the isolation of analytical queries and applications from the technological details imposed by the sources and (b) accommodates syntactic evolution from the sources. The introduced ontology builds upon known ideas from ontology-based data access (OBDA) research [19], and includes two layers in order to provide analysts with an integrated and format-agnostic view of the sources. We exploit this structure to handle the evolution of source schema via semi-automated transformations on the ontology upon service releases. Our approach is based on well known Semantic Web technologies, specifically RDF, which contrary to other schema definition languages (e.g., XSD) enable (a) reutilization of existing vocabularies, (b) self-description of data, and (c) publishing such data on the web [3]. Our contributions can be summarized as follows:

- We introduce a structured ontology, discussed in Section 2, that allows to model and integrate situational data from multiple data providers. As an add-on, we take advantage of RDF's nature to provide semantics by means of Linked Data.
- We present a method that handles schema evolution on the sources, see Section 3. In practice, we flexibly accommodate source changes by only applying changes to the ontology dismissing the need to change the analytical processes logic.
- We assess our method by performing a functional evaluation w.r.t. the results of RESTful API evolution studies. The evaluation discussed in Section 4 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.

Our discussion is complemented by reviewing related work in Section 5 and open issues for future work in Section 6.

2. BIG DATA INTEGRATION ONTOLOGY

In this section, we present the Big Data Integration ontology (BDI), the metadata artifact enabling to systematically govern the data ingestion and analysis process. Its goal is to model and integrate, in a machine-readable format, semi-structured data while preserving data independence regardless of the source formats or schema. To this end, it is divided into two levels linked to each other by means of mappings. The global level provides a unified schema for querying as well as relevant metadata about the attributes, while the source level deals with the physical details of each data source.

2.1 Preliminaries

In this work, we present the BDI ontology as an instantiation of the theoretical data integration (DI) framework by Lenzerini [9]. Shortly, a DI system $\mathcal{I}$ is formalized as a triple $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, which respectively represent the global schema, the source schema and the mappings. $\mathcal{S}$ describes the structure of the sources, while $\mathcal{G}$ provides an integrated view on which queries will be posed. This is achieved through $\mathcal{M}$, a set of assertions $q_{\mathcal{G}} \rightsquigarrow q_{\mathcal{S}}$ or $q_{\mathcal{S}} \rightsquigarrow q_{\mathcal{G}}$, being $q_{\mathcal{G}}$ and $q_{\mathcal{S}}$ queries over $\mathcal{G}$ and $\mathcal{S}$, respectively. The mappings follow the local as view (LAV) approach when for each element s of $\mathcal{S}$, assertions are of the form $s \rightsquigarrow q_{\mathcal{G}}$. Conversely, they follow the global as view (GAV) approach when for each element g of $\mathcal{G}$, assertions are of the form $g \rightsquigarrow q_{\mathcal{S}}$.

A BDI ontology $\mathcal{O}$ is defined as a 3-tuple $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ of RDF graphs. The rationale behind the BDI ontology is

[4]As measured by NLTK Text Classification (http://text-processing.com/demo/sentiment)

[5]https://dev.twitter.com/rest/reference/get/search/tweets

to provide a metadata model to systematically annotate ingested situational data (i.e., events) from the sources (i.e., by means of $\mathcal{S}$), while allowing data analysts to query a static integrated schema that creates an abstraction layer of the underlying physical details (i.e., by means of $\mathcal{G}$). The key aspect is that it allows ingesting and storing data in its original source format, given that $\mathcal{S}$ is an accurate abstraction of the events, and relies on query rewriting techniques (i.e., by means of $\mathcal{M}$) to translate the queries to the corresponding format. All this is possible, thanks to the extensibility of RDF which enables to enrich $\mathcal{G}$ and $\mathcal{S}$ with the necessary metadata such as constraints, data types or data formats.

Our approach (see Figure 1) introduces the figure 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) incorporating to $\mathcal{S}$ the triple-based representation of the schema of newly incoming events (E_i) produced by APIs, and (b) make such data available for data analysts to query (Q_i) by creating mappings from $\mathcal{S}$ to $\mathcal{G}$. In the following sections we elaborate on each level that compose $\mathcal{O}$ presenting their metadata model, as well as their mapping to the case study.

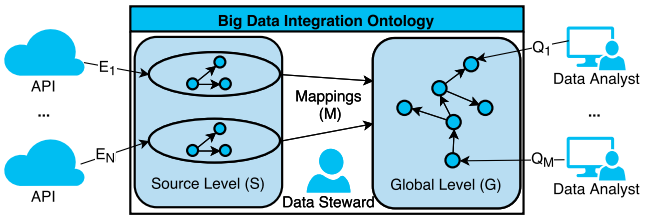

Figure 1: High-level overview of our approach

2.2 Global Level

The global level $\mathcal{G}$ 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 purpose is similar to the global schema in $\mathcal{I}$, and 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. Concepts can be linked by means of domain-specific object properties with `rdfs:domain` and `rdfs:range`. The link between a concept and its set of features is achieved via `G:hasFeature`. Additionally, we enrich such constructs with new semantics to aid the data management and analysis phases. In this paper, we narrow the scope to two properties widely used in data integrity management, namely integrity constraints and data types for features. Such information can help to an easier development of the processing logic by assuring data consistency.

Code 2 provides the triples that compose $\mathcal{G}$ in Turtle RDF notation[6]. It contains the main metaclasses (using prefix `G`[7] 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 integrity constraints and data types on features, respectively linked using `G:hasConstraint` and `G:hasDatatype`. Following the same LD philosophy, we reuse the `rdfs:Datatype` vocabulary 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`[8]). Finally, note that here we focus on non-complex data types, however our model can be easily extended to include complex types [6].

```
@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 G: <http://www.BDIOntology.com/global/> .
<http://www.BDIOntology.com/global/> rdf:type owl:Ontology .

G:hasFeature rdf:type owl:AnnotationProperty .
G:hasConstraint rdf:type owl:AnnotationProperty .
G:hasDatatype rdf:type owl:AnnotationProperty .

G:IntegrityConstraint rdf:type owl:Class .
G:Feature rdf:type owl:Class ;
                G:hasConstraint G:IntegrityConstraint ;
                G:hasDatatype rdfs:Datatype .
G:Concept rdf:type owl:Class ;
                G:hasFeature G:Feature .
```

Code 2: Metadata model for $\mathcal{G}$ in Turtle notation

Case Study.

Figure 2 depicts the instantiation of $\mathcal{G}$ in SUPERSEDE. The color of the elements depicts instances of the data model to the metadata model (i.e., `rdf:type` links). 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`). Regarding integrity constraints, we provide three instantiations: `sup:CurrentTimeIfEmpty` specifying a default value of the current date in the case of missing value; `sup:NotNull` ensuring that the linked features will always have a value and `sup:Length-140` guaranteeing that the length of the tweet is no longer than 140 characters.

2.3 Source Level

The source level $\mathcal{S}$ has the same purpose as the source schema in $\mathcal{I}$, to model the data ingested from the sources. By maintaining certain properties describing the source format we allow the modeling of semi-structured data. Code 3 depicts the metadata model for $\mathcal{S}$ in Turtle RDF notation (using prefix `S`[9] as main namespace). As done for other modeling languages [15], we define the concept `S:Event` which models different ingested types of event (e.g., a JSON document), one for each API. To support historical analysis of stored data, as well as API evolution, events can produce different schema versions which in turn consist of, possibly shared, sets of attributes. In the context of this paper, we focus on the ingestion of JSON data, hence we define the concepts `S:EmbeddedObject`, `S:Array` and `S:Attribute` with their respective links. In addition, schema versions are enriched with the physical format of the data source, leveraging on class `dcat:mediaType`. Such class, as part of the Data Catalog Vocabulary[10], offers a wide range of data formats[11] aiding in the definition of the specific parsing mechanism.

[6] https://www.w3.org/TR/turtle
[7] http://www.BDIOntology.com/global
[8] http://www.w3.org/2001/XMLSchema
[9] http://www.BDIOntology.com/source
[10] https://www.w3.org/TR/vocab-dcat
[11] https://www.iana.org/assignments/media-types/media-types.xhtml

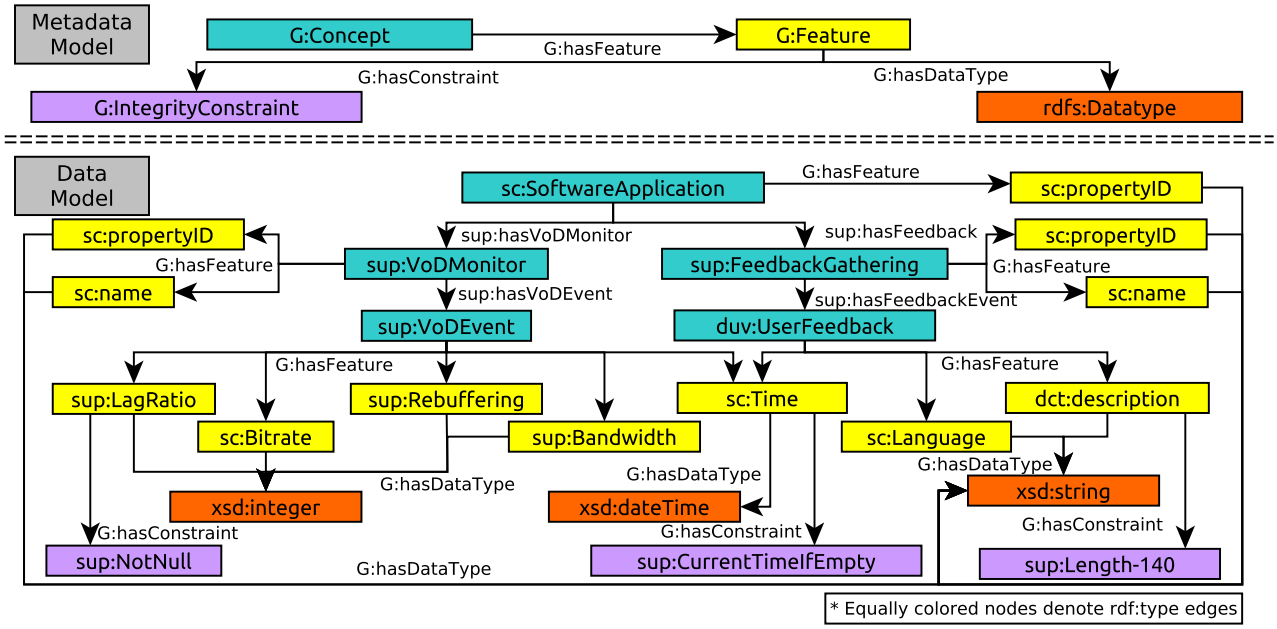

Figure 2: Instantiation of $\mathcal{G}$ for the SUPERSEDE case study

```
@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 dct: <http://purl.org/dc/terms/> .
@prefix dcat: <http://www.w3.org/ns/dcat#> .
@prefix S: <http://www.BDIOntology.com/source/> .
<http://www.BDIOntology.com/source/> rdf:type owl:Ontology .

dct:format rdf:type owl:AnnotationProperty .
S:hasAttribute rdf:type owl:AnnotationProperty .
S:hasArray rdf:type owl:AnnotationProperty .
S:hasEmbeddedObject rdf:type owl:AnnotationProperty .

S:Attribute rdf:type owl:Class .
S:EmbeddedObject rdf:type owl:Class ;
          S:hasEmbeddedObject S:EmbeddedObject ;
          S:hasArray S:Array ;
          S:hasAttribute S:Attribute .
S:Array rdf:type owl:Class ;
          S:hasEmbeddedObject S:EmbeddedObject ;
          S:hasArray S:Array ;
          S:hasAttribute S:Attribute .
S:SchemaVersion rdf:type owl:Class ;
          dct:format dcat:mediaType ;
          S:hasEmbeddedObject S:EmbeddedObject ;
          S:hasArray S:Array ;
          S:hasAttribute S:Attribute .
S:Event rdf:type owl:Class ;
          S:produces S:SchemaVersion .
```

Code 3: Metadata model for $\mathcal{S}$ in Turtle notation

As $\mathcal{S}$ is an accurate representation of the event's structure, we can leverage on schema languages, such as JSON Schema [18], for its automatic construction. However, this is not always available, thus in this paper we advocate for the automatic construction of $\mathcal{S}$ on the basis of a reference sample dataset (i.e., a JSON document). Algorithm 1 depicts the recursive process of generating the triples describing such schema. Its inputs are the source level to populate, the sample dataset and the parent's IRI. An IRI is a unique string identifier of an element of the ontology that is expressed as an absolute path from the root of the ontology to the element on hand. Such last parameter is used to isolate attributes from different events within the same namespace. In the first iteration, the parent element is the event's IRI, and in successive recursive calls are the JSON keys for arrays or embedded objects. For instance, the `id_str` key from Code 1 will be stored with the IRI `sup:Twitter/user/id_str`.

Algorithm 1 Extract JSON Recursively

Pre: $\mathcal{S}$ is the source level, J is the sample JSON dataset and *parent* the parent's node IRI in $\mathcal{S}$
Post: $\mathcal{S}$ contains the triples representing the structure of the contents of J as children of *parent*

```
1:  function ExtractRec(S, J, parent)
2:    for each (k, v) ∈ J do
3:      IRI = parent+"/"+k
4:      S ∪= ⟨IRI, "rdf:type", getClass(J, k)⟩
5:      S ∪= ⟨parent, getLink(J, k), IRI⟩
6:      if getClass(J, k) = JSONObject.class then
7:        extractRec(S, J(k), IRI)
8:      else if getClass(J, k) = JSONArray.class then
9:        extractRec(S, J(k)[0], IRI)
10:     end if
11:   end for
12:   return S
13: end function
```

Algorithm 1 makes use of the auxiliary functions getClass and getLink. The purpose of the former is to return the specific URIs for the JSON key on hand (`S:EmbeddedObject`, `S:Array` or `S:Attribute`). Likewise, the purpose of the latter is to return the respective edge (`S:hasEmbeddedObject`, `S:hasArray` or `S:hasAttribute`). Finally, note that we assume arrays with uniform structures, thus when exploring the elements of an array it is only necessary to check the first one (see line 9). In Section 3.2 we further elaborate on the fully automated construction of $\mathcal{S}$ for a new version release.

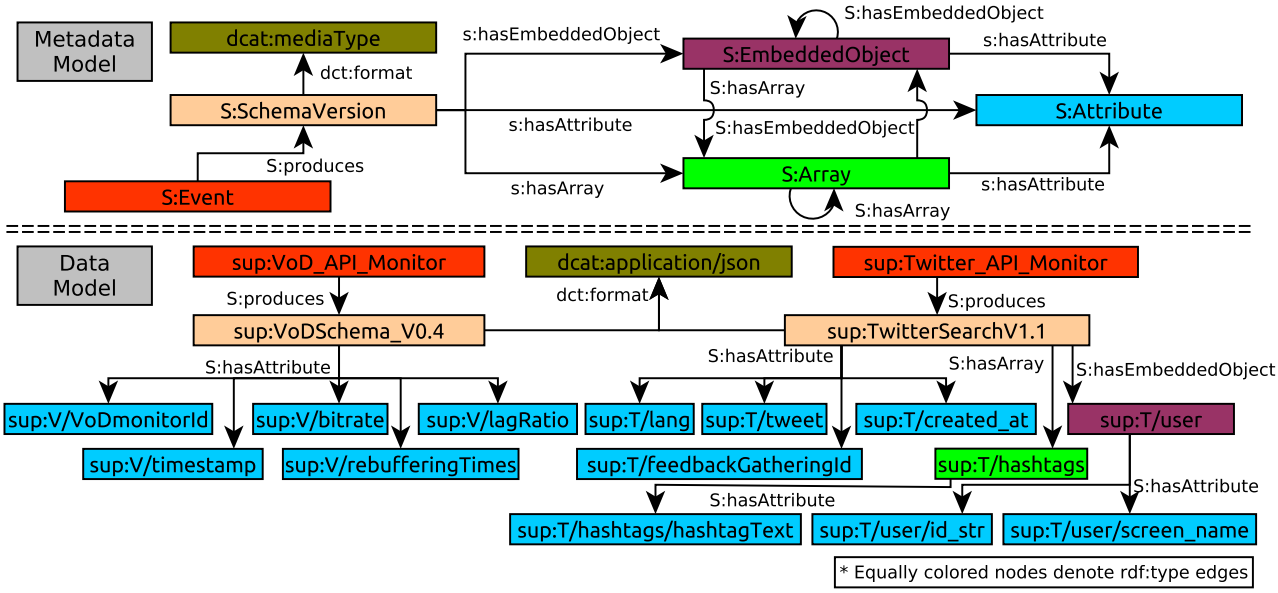

Figure 3: Instantiation of $\mathcal{S}$ for the SUPERSEDE case study

Case Study.

Figure 3 shows the instantiation of $\mathcal{S}$ in SUPERSEDE. The two sources produce the events in Code 1. Elements in $\mathcal{S}$ have been obtained using Algorithm 1, invoking it as ExtractRec($\emptyset$, `{...}`, `"sup:Twitter_API_Monitor"`) for the Twitter monitor, and similarly for the VoD. We simplify IRIs as V and T, respectively for VoD and Twitter monitors.

2.4 Mappings

In this subsection, we discuss how the mappings link the different levels composing $\mathcal{O}$. Our approach uses two types of mappings, those linking JSON events with their triple-based schema representation, and those linking the global and source levels. On one hand, the former are implicitely linked by the JSON keys and the IRIs, as constructed by Algorithm 1, and thus we have an accurate one-to-one mapping. On the other hand, the latter consists of the triplets present in $\mathcal{M}$. In order to obtain clean semantics for the mappings, we limit them to be in the form $t_{\mathcal{S}} \rightsquigarrow t_{\mathcal{G}}$ (i.e., triples $\langle t_{\mathcal{S}}, \texttt{S:mapsTo}, t_{\mathcal{G}} \rangle$), this allows to better accommodate source evolution. Note we restrict that each source element will have a mapping to exactly one global element in order to simplify the query rewriting process to unfolding [8]. To forestall any possible criticism, we would like to clarify that constraining source elements to map to exactly one global element is not overrestrictive or oversimiplistic, but rather it provides clean semantics for our sources. What would be problematic would be the inverse (constraining global elements to single providers), but, as we show in the following subsection, there is no such problem. With such final construct, we can depict the complete metadata model for $\mathcal{O}$ in Figure 4.

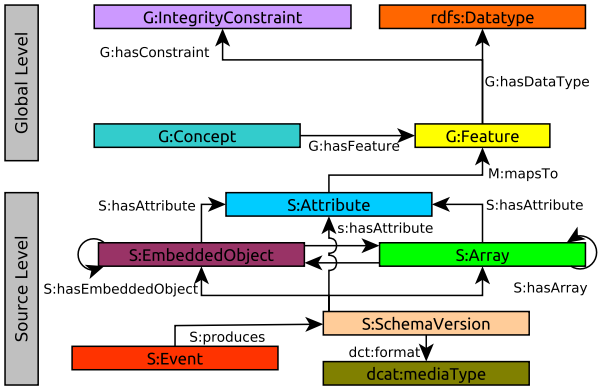

Figure 4: Complete metadata model for $\mathcal{O}$

2.5 Querying Via the Ontology

As previously mentioned, queries will be issued to the elements of $\mathcal{G}$ and rewritten (unfolding the mappings) to the JSON events in a two-phase manner. Firstly, by issuing a SPARQL query to $\mathcal{M}$, elements in $\mathcal{S}$ that are associated to the elements of $\mathcal{G}$ in the user query are devised. Note that the SPARQL query is traversing a single path, and hence has low complexity. Note that this first phase may require the data analyst to resolve potential ambiguities. This is due to the fact that a global feature may have many source attributes from different events (see `sc:Time` in Figure 5). The second phase consists of translating the query to an specific programming language to parse the source format of the event on hand (annotated with `dcat:mediaType`). Mappings between $\mathcal{S}$ and the events are depicted in the IRI of attributes, yielding accurate one-to-one transformations. Many proposals exist for ontology-based querying such as Ontop [4], an OBDA system providing access to relational databases through a domain ontology.

Case Study.

Figure 5 depicts the complete instantiation for $\mathcal{O}$ in SUPERSEDE. To ensure readability, internal classes are omitted and only the core ones are shown. In SUPERSEDE,

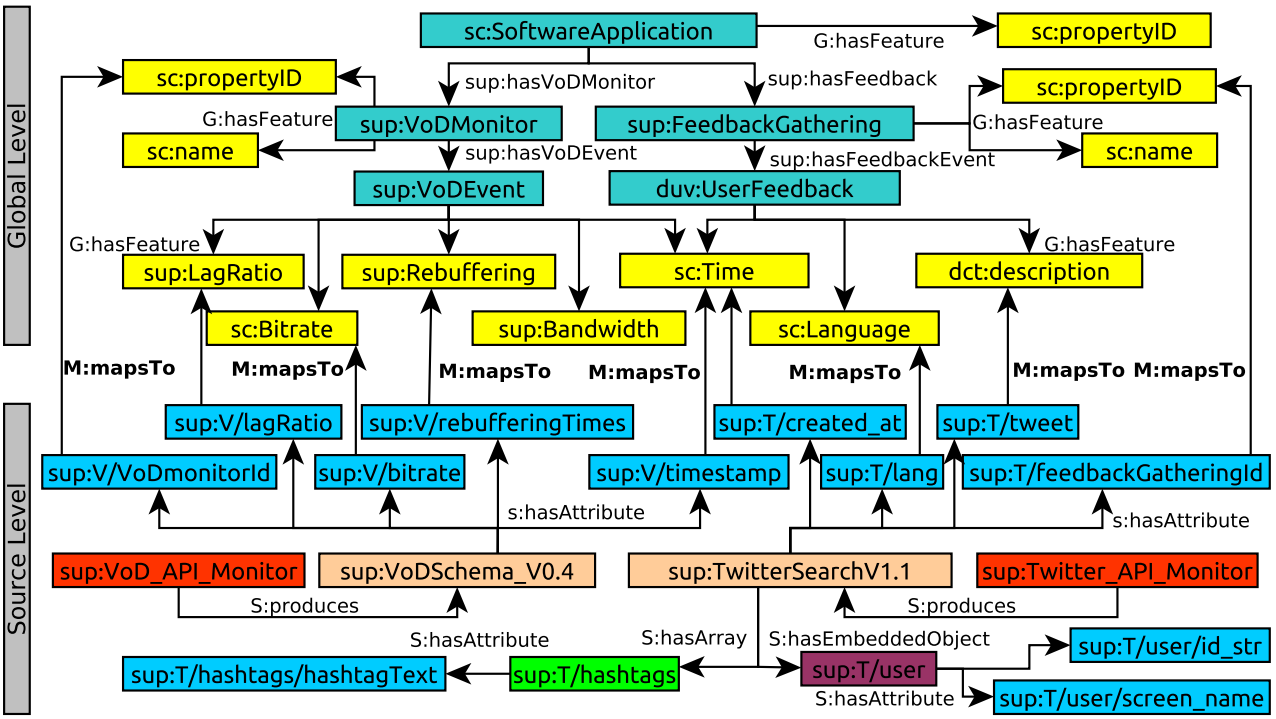

Figure 5: Instantiation of $\mathcal{O}$ for the SUPERSEDE case study

monitors push data to Apache Kafka, a distributed message queue that distinguishes incoming streams using *topics*. A stream is assigned a topic for each combination $\langle Event, SchemaVersion \rangle$. In Figure 6 we depict the *QoE* integrated analysis (in the form of an SQL-like query) and the unfolding to resolve it using the BDI ontology.

```
SELECT AVG((1-V.sup:LagRatio/100) *
  sentiment(UF.dct:description))
FROM sup:VoDEvent V, duv:UserFeedback UF
WHERE V.sc:Time = today() && UF.sc:Time = today()
```

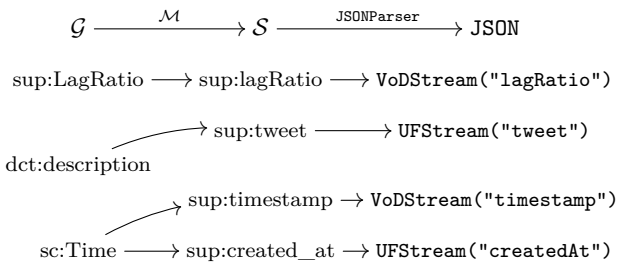

Figure 6: Example of SQL-like query and its 2-phase unfolding process

3. HANDLING EVOLUTION

In this section, we present how the BDI ontology accomodates evolution of situational data. Chapin et al. distinguish between three main areas in software evolution: code, software and customer-experienced functionality [5]. Specific studies REST API evolution [10, 24] have concluded that most of such changes occur in the code, and thus in the structure of incoming events. 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.

3.1 Releases

In Section 2.1, 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 data analysts. Now, the goal is to shield such analytical processes, implemented on top of $\mathcal{G}$, so that they do not crash upon new API version releases. In other words, we need to adapt $\mathcal{S}$ to such schema evolution in the events, so that $\mathcal{G}$ is not affected. To this end, we introduce the notion of *release*, the construct indicating which elements from the new event (Evt) will be used for analysis and how do they link the global level. Formally, a release R is defined as a 3-tuple $\langle J, G, \theta \rangle$, where: $J \subset Evt$ is the set of source schema elements made available for analysis, $G \subset \mathcal{G}$ is the set of global elements from $\mathcal{O}$ related to the new release, and $\theta = J \mapsto G$ is a bijective function mapping the JSON elements with those in the global level.

R must be created by the data steward upon new releases. Several approaches can aid this process, such as PARIS [22] which uses probabilistic methods to align and match RDF ontologies. In the following subsection, we discuss how R serves as input to the algorithm that will automatically adapt $\mathcal{O}$, ensuring the correct functioning of queries.

Case Study.

For the case of Twitter feedback monitors, the release R would be depicted as $J = \{$`created_at`, `lang`, `tweet`, `hashtagText`, `id_str`, `feedbackGatheringId`, `screen_name`$\}$,

G = {sc:Time, sc:Language, sc:name, dct:description, sc:propertyID} and θ = { lang ↦ sc:Language, created_at ↦ sc:Time, feedbackGatheringId ↦ sc:propertyID, tweet ↦ dct:description }.

3.2 Release-based Ontology Evolution

As mentioned above, changes in the event's elements require reacting in order to avoid queries to crash. Furthermore, the ultimate goal is to provide such adaptation in an automated way dismissing the need of the developer's interaction. To this end, Algorithm 2 applies the necessary changes to adapt the BDI ontology $\mathcal{O}$ w.r.t a new release R. It starts registering the event endpoint, in case it is new (line 4), and the new schema version to further link them (lines 7 and 8). Then, the JSON ($R(J)$) is used to create a triple-based representation of its schema (calling Algorithm 1, in line 10). With such, it is possible to iterate on its elements and check their existence in the current source level $\mathcal{O}(\mathcal{S})$. Given the way IRIs for attributes are constructed in Algorithm 1, we can ensure that only attributes from the same event will be reused within subsequent versions. This helps to maintain a low growing rate of $\mathcal{O}(\mathcal{S})$, as well as avoiding potential semantic differences that would cause mappings to different global elements. Finally, by iterating on the partial function ($R(\theta)$) the mappings to the global level are generated (line 19). Note, the usage of the auxiliary method FINDIRI in line 17. Given a textual JSON key, it returns its fully qualified IRI in the source level.

Algorithm 2 Adapt to Release

Pre: $\mathcal{O}$ BDI ontology, R new release, Evt event and V new version

Post: $\mathcal{O}$ is adapted w.r.t R

1: **function** NEWRELEASE($\mathcal{O}$, R, Evt, V)
2: Evt_{uri} = "S:Event/"+Evt
3: **if** $Evt_{uri} \notin \mathcal{O}(\mathcal{S})$ **then**
4: $\mathcal{O}(\mathcal{S}) \cup= \langle Evt_{uri}, \texttt{"rdf:type"}, \texttt{"S:Event"} \rangle$
5: **end if**
6: V_{uri} = "S:SchemaVersion/"+V
7: $\mathcal{O}(\mathcal{S}) \cup= \langle V_{uri}, \texttt{"rdf:type"}, \texttt{"S:SchemaVersion"} \rangle$
8: $\mathcal{O}(\mathcal{S}) \cup= \langle Evt_{uri}, \texttt{"S:produces"}, V_{uri} \rangle$
9: $\mathcal{O}(\mathcal{S}) \cup= \langle V_{uri}, \texttt{"dct:format"}, \texttt{"dcat:appl/json"} \rangle$
10: $\mathcal{S}_{new}$ = EXTRACTREC($\mathcal{O}(\mathcal{S})$, $R(J)$, V_{uri})
11: **for each** $s \in \mathcal{S}_{new}$ **do**
12: **if** $s \notin \mathcal{O}(\mathcal{S})$ **then**
13: $\mathcal{O}(\mathcal{S}) \cup= s$
14: **end if**
15: **end for**
16: **for each** $(j, g) \in R(\theta)$ **do**
17: j_{iri} = FINDIRI(j, $\mathcal{S}_{new}$)
18: g_{iri} = "G:Feature/"+g
19: $\mathcal{O}(\mathcal{M}) \cup= \langle j_{iri}, \texttt{"M:mapsTo"}, g_{iri} \rangle$
20: **end for**
21: **end function**

Case Study.

Let us assume that the Twitter feedback monitor releases a new simplified version of the schema in XML format (see the sample XML event in Code 4).

The new release R can be depicted as J = {created_at, locale, tweetText, feedbackGatheringId}, G = {sc:Time, sc:Language, sc:name, dct:description, sc:propertyID}, θ = { locale ↦ sc:Language, created_at ↦ sc:Time, tweetText ↦ dct:description, feedbackGatheringId ↦ sc:propertyID }. Figure 7 depicts the resulting BDI ontology $\mathcal{O}$ after running Algorithm 2 with such input. In the bottom left, we depict the old source level, while in the bottom right the evolved one. In the top, we depict the global level, which is not changed. Elements with the same color depict edges of type S:mapsTo from the source to the global level. Finally, for readability, URIs have been simplified and do not strictly adhere to those generated in Algorithm 2.

```
<Twitter_API_Monitor_Simple>
  <feedbackGatheringId>28</feedbackGatheringId>
  <locale>en</locale>
  <tweetText>I no longer see the loading symbol</tweetText>
  <created_at>1475010425</created_at>
</Twitter_API_Monitor_Simple>
```

Code 4: XML event for the new Twitter version

4. EVALUATION

In this section, we present the evaluation results on our approach. We provide a functional evaluation on evolution management and discuss performance issues.

4.1 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 [24]. 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. Throughout the paper, we assumed the existence of a set of monitoring tools, from now on we refer to them as monitors, in charge of bridging the communication between the API providers and the Big Data architecture [16]. Besides pulling data from the API endpoints, they also enrich the response with additional information (e.g., the feedbackGatheringId attribute in SUPERSEDE). All functionalities related to fetch data from the API provider (e.g., authentication or HTTP query parametrization) are delegated to them. With this in mind, we provide the list of changes per level and indicate the component responsible of it (i.e., either *Monitor* or *BDI Ont.*). For those functionalities managed by the BDI ontology we later discuss their specific rationale.

API-level Changes.

Those changes concern the whole of an API and it can be observed either because a new endpoint is incorporated (e.g., a new social network in the SUPERSEDE use case) or to update all methods concerning one provider. Table 1 depicts the API-level change breakdown and the component responsible to handle it.

Adding or changing a response format at API level consists of, for each event from such provider, registering a new release with this format. Regarding the deletion of a response format, it does not require any action, 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 $\mathcal{O}$.

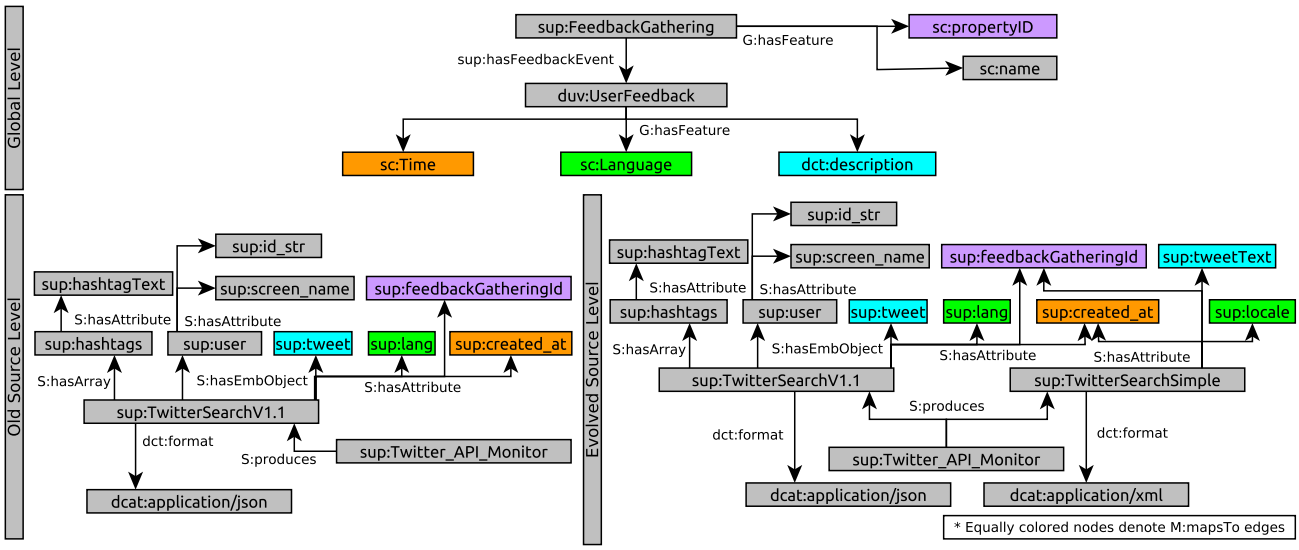

Figure 7: Old and evolved $\mathcal{S}$ for the SUPERSEDE case study, $\mathcal{G}$ does not change

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

Table 1: API-level changes dealt by monitors or BDI ontology

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 2 summarizes the method-level change breakdown and the component responsible to handle it.

Method-level Change	Monitor	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 2: Method-level changes dealt by monitors or BDI ontology

Those changes have more overlapping with the monitors 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:Event` and thus, adding a new one consists on declaring a new release and running Algorithm 2. Renaming a method requires renaming the Event instance. As before, a removal does not entail any action with the goal 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 3 depicts such changes and the component in charge of handling it.

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

Table 3: Parameter-level changes dealt by monitors or BDI ontology

Similarly to the previous level, some parameter-level changes are managed by both monitors and the ontology. This is caused by the ambiguity of the change statements, and hence we might consider both URL query parameters and JSON response parameters. 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 event on hand. Specifically, the changes will be depicted in the release dataset $R(J)$.

4.2 Industrial Applicability

After functionally validating that the BDI ontology can handle all types of API evolution, next we aim to study how these frequent changes occur in real-world APIs. For this purpose we study the results from [10], a similar study as the one in the previous subsection. In a nutshell, [10] presents 16 change patterns that frequently occur in the evolution of 5 widely used APIs. With such information, we can show the number of API changes per API that could be accommodated

API Owner	#Changes Monitor	#Changes Ontology	#Changes Monitor&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 4: Number of changes per API and percentage of partially and fully accommodated changes by $\mathcal{O}$

by the BDI ontology. We summarize the results in Table 4. As before, we distinguish between changes concerning (a) the monitors, (b) the ontology and (c) both monitors 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 monitors) and those fully accommodated (changes only concerning the ontology).

Our results depict 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 enables to tackle on average 71.62% of changes.

4.3 Performance Evaluation

In this final evaluation we are concerned with performance-wise 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[12]. 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.

Methodology.

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 changes. 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 [14]).

Results.

The barcharts in Figure 8 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 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.

With such analysis we conclude that major version changes entail a steep 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 cummulative number of triples after each release. For a practically stable amount of minor release versions we obtain a linear, stable growth in $\mathcal{S}$. This guarantees that querying $\mathcal{O}$ for unfolding 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.

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

5. RELATED WORK

As we discussed in Section 1, in this paper we are concerned with challenges related to *Variety* and *Variability* in Big Data ecosystems. Thus, in this section we independently study the related work for such research lines.

Governance of Big Data Ecosystems.

Lots of efforts are nowadays being put by the research community on the governance of Big Data ecosystems. Project Constance [7] aims to cover the lack of semantics in data lakes. Specifically, its metadata management system GEMMS [20] creates a unified metamodel for raw data and allows querying the data with rewriting methods such as ours. Our work differs from Constance in the manner that we do not narrow the scope to data lakes (e.g., batch analysis) but also to real-time data in the forms of data streams. We additionally adopt techniques to accommodate schema evolution. Other approaches have also identified the need to propose a *curated* data lake enriched with semantics [23]. However, matters on streaming data and evolution are overlooked as before.

API and Database Evolution.

In previous sections, we have cited relevant works on RESTful API evolution [24, 10]. They provide a catalog of changes, however they do not provide any approach to systematically deal with them. Other similar works, such as [25], empirically study API evolution aiming to detect its healthiness. If we look for approaches that automatically deal with evolution, we must shift the focus to the area of database schema evolution. Such works, however, are mostly focused on relational databases [21, 11]. 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.

6. 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 metadata-driven approach. The presented algorithms 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 monitors and the ontology. There are many interesting future directions. A prominent one is to extend

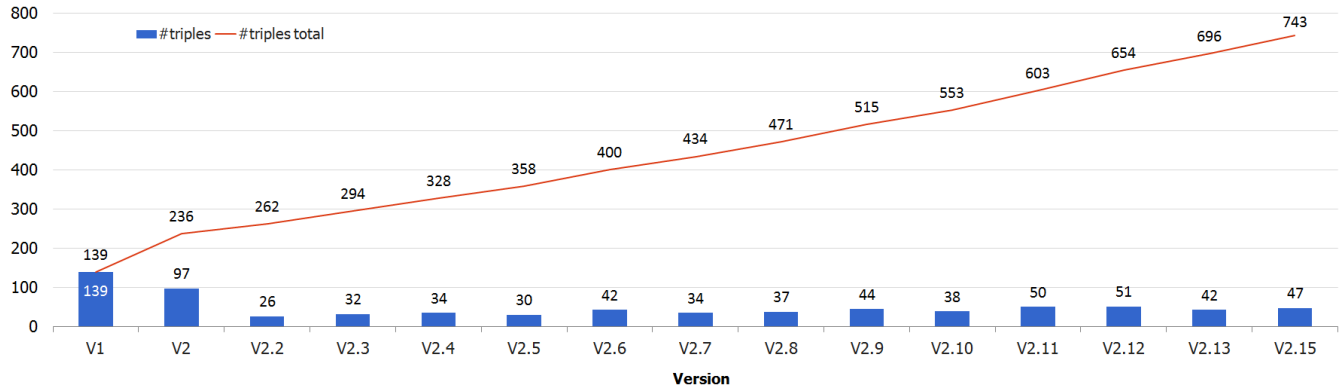

Figure 8: Growth in number of triples for $\mathcal{S}$ per release in Wordpress API

the ontology with richer constructs to semi-automatically adapt to unanticipated schema evolution.

7. ACKNOWLEDGEMENTS

This work has been partly supported by the H2020 SUPERSEDE project, funded by the EU Information and Communication Technologies Programme (num. 644018), and the GENESIS project, funded by the Spanish Ministerio de Ciencia e Innovación (num. TIN2016-79269-R).

8. REFERENCES

[1] A. Abelló. Big Data Design. In *DOLAP*, pages 35–38, 2015.

[2] 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.

[3] C. Bizer, T. Heath, and T. Berners-Lee. Linked Data - The Story So Far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[4] 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):471–487, 2017.

[5] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W. Tan. Types of Software Evolution and Software Maintenance. *Journal of Software Maintenance*, 13(1):3–30, 2001.

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

[7] R. Hai, S. Geisler, and C. Quix. Constance: An Intelligent Data Lake System. In *SIGMOD*, pages 2097–2100, 2016.

[8] 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.

[9] M. Lenzerini. Data Integration: A Theoretical Perspective. In *PODS*, pages 233–246, 2002.

[10] J. Li, Y. Xiong, X. Liu, and L. Zhang. How Does Web Service API Evolution Affect Clients? In *ICWS*, pages 300–307, 2013.

[11] 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.

[12] V. Markl. Situational Business Intelligence. In *BIRTE*, 2008.

[13] N. Marz and J. Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications Co., 1st edition, 2015.

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

[15] A. Olivé and R. Raventós. Modeling Events as Entities in Object-oriented Conceptual Modeling Languages. *Data Knowl. Eng.*, 58(3):243–262, 2006.

[16] M. Oriol, X. Franch, and J. Marco. Monitoring the Service-based System Lifecycle with SALMon. *Expert Syst. Appl.*, 42(19):6507–6521, 2015.

[17] C. Pautasso, O. Zimmermann, and F. Leymann. Restful Web Services vs. "Big" Web Services: Making The Right Architectural Decision. In *WWW*, pages 805–814, 2008.

[18] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoc. Foundations of JSON Schema. In *WWW*, pages 263–273, 2016.

[19] 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.

[20] C. Quix, R. Hai, and I. Vatov. GEMMS: A Generic and Extensible Metadata Management System for Data Lakes. In *CAiSE*, pages 129–136, 2016.

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

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

[23] I. Terrizzano, P. M. Schwarz, M. Roth, and J. E. Colino. Data Wrangling: The Challenging Yourney from the Wild to the Lake. In *CIDR*, 2015.

[24] S. Wang, I. Keivanloo, and Y. Zou. How Do Developers React to RESTful API Evolution? In *ICSOC*, pages 245–259, 2014.

[25] A. V. Zarras, P. Vassiliadis, and I. Dinos. Keep Calm and Wait for the Spike! Insights on the Evolution of Amazon Services. In *CAiSE*, pages 444–458, 2016.