Ontology-Based Generation of Data Platform Assets

Vincenzo De Leo, Gianni Fenu
Department of Math and Computer Science
University of Cagliari
Cagliari, Italy
fenu@unica.it

David Greco, Nicolò Bidotti, Paolo Platter
Big Data Laboratory
AgileLab s.r.l.
Rome, Italy
{david.greco,nicolo.bidotti,paolo.platter}@agilelab.it

Enrico Motta
Knowledge Media Institute
Open University
Milton Keynes, UK
enrico.motta@open.ac.uk

Andrea Giovanni Nuzzolese
Institute of Cognitive Sciences and Technologies
National Council of Research
Bologna, Italy
andreagiovanni.nuzzolese@cnr.it

Francesco Osborne
Knowledge Media Institute
Open University
Milton Keynes, UK
francesco.osborne@open.ac.uk

Diego Reforgiato Recupero
Department of Math and Computer Science
University of Cagliari
Cagliari, Italy
diego.reforgiato@unica.it

***Abstract*—The design and management of modern big data platforms are extremely complex. It requires carefully integrating multiple storage and computational platforms as well as implementing approaches to protect and audit data access. Therefore, onboarding new data and implementing new data transformation processes is typically time-consuming and expensive. In many cases, enterprises construct their data platforms without a clear distinction between logical and technical concerns. Consequently, these platforms lack sufficient abstraction and are closely tied to particular technologies, making the adaptation to technological evolution very costly. This paper illustrates a novel approach to designing data platform models based on a formal ontology that structures various domain components into an accessible knowledge graph. We also describe the preliminary version of AGILE-DM, a novel ontology that we built for this purpose. Our solution is flexible, technologically agnostic, and more adaptable to changes and technical advancements.**

***Index Terms*—Data platform, Ontology, Multi-level architecture, Knowledge Graphs**

I. Introduction

The advent of big data has brought forth many challenges in terms of data management, storage, analysis, and modeling.

Designing, building, and managing a modern data platform is extremely complex. These processes involve the integration of multiple storage and computational platforms and the implementation of a complex mechanism for protecting data access and auditing it. Moreover, a data platform should support the definition of policies and constraints to keep it constantly aligned with the business and technical requirements. Given this complexity, onboarding new data and implementing new data transformation processes is typically time-consuming and expensive. Additionally, enterprises often construct their data platforms without a robust engineering process or a clear distinction between logical and technical concerns. Consequently, these platforms lack sufficient abstraction and are closely tied to particular technologies, making the adaptation to technological evolution very costly. Furthermore, the absence of formal abstraction or logical modelling in data platforms hinders the introduction of automation, thereby increasing the time required to deliver new data and execute data transformations. Such a poor level of automation is quite common and ultimately results in the platforms' inability to keep pace with business demands. It is increasingly evident that supporting the governance of a modern data platform needs a more flexible catalog capable of containing both asset definitions and metadata, where an asset can be any resource composing a data platform. For example, a data mesh architecture consists of numerous assets, typically data products containing input and output ports. Usually, this structure is spread out in various places, which makes it difficult to manage and update. Our approach represents all these parts in one easy-to-use knowledge graph.

Although many different definition languages, data formats, schemas, and specifications have been provided [1], [2], to the best of our knowledge, no one has ever proposed an approach to dynamically create a logical model.

In order to address this challenge, we present a methodology for building data platform models as knowledge graphs that are structured according to a preliminary version of the Agile Data Management Ontology (AGILE-DM), which we specifically designed for describing big data platforms. In the last few years, knowledge graphs have become prominent for representing structured data in a meaningful way [3]. They efficiently capture complex relationships between entities and attributes, providing a format that machines can read, useful for various intelligent services [4]. The knowledge graph based on AGILE-DM simplifies the data platform's management, querying, and updating. Therefore, our solution is flexible, technologically agnostic, and more adaptable to changes and technical advancements.

A domain ontology, such as AGILE-DM, allows practitioners to describe how the data are organized, establish policies governing the mapping of logical entities to their technological counterparts, and specify the ongoing constraints the data platform must adhere to. AGILE-DM has been developed with a multi-level architecture that consists of four distinct layers.

The first layer provides the language for defining arbitrary types, providing a stable foundation. The second layer

describes a set of traits that formalize the behaviors and relationships of the asset types. The third layer contains all the dynamically defined user types. The fourth and final layer comprises the instances of user-defined asset types. It will be generated by a software platform on top of the types and traits defined in the other three layers. As such, this ontology is very flexible and can be used to describe all the possible assets composing a data platform. In summary, the main contributions of this paper are the following:

- we design an approach to dynamically create a data platform logical model;
- we define a multi-level ontology that includes data types and traits to support the dynamic creation of data platform models.

II. Related Work

In the last decades, a lot of work has been done to define credible models of knowledge, encompassing different subfields of the Semantic Web such as Knowledge Representation and Knowledge Engineering [3]. However, few works have proposed solutions for defining ontological artefacts for modelling and managing data platform assets in big data scenarios [5] in which the following characteristics hold: (i) massive volume; (ii) high velocity for the data management; (iii) multiple varieties due to the high heterogeneous data sources. Many ontology-based solutions in literature focused on specific aspects of big data management. In the context of ontology-based data access (OBDA) [6], a significant amount of work has been proposed in the literature. The literature mainly focuses on heterogeneity issues in data sources by levering ontologies by solving a data integration problem [2]. Some solutions include data integration systems following the local-as-view (LAV), global-as-view paradigms (GAV), and global and local-as-view (GLAV) [7]. These approaches are enabled by different techniques such as graph-based traversal [8], [9], and distributed and federated query processing [10]–[13]. Similarly, a number of ontologies describe the management of data and can be used in the context of big data platforms. Those ontologies typically focus on provenance, encompassing various transformations, analyses, and interpretations applied to the data. A reference example in the Semantic Web community is the PROVO ontology[1] [1], which can be used to represent provenance information generated by different systems. DCAT[2] is an RDF vocabulary designed to facilitate interoperability between data catalogs published on the Web. There are also ontologies that serve as the foundation for systems dedicated to enhancing situation awareness and decision-making based on data. KIDS (Knowledge Intensive Data System) [14] is a notable example of this family of ontologies, which relies on the Observe, Orient, Decide, Act loop (OODA) [15] and aims at representing both the data hierarchy (consisting of Facts, Perceptions, Hypotheses, and Directives) that elucidates the process of generating action directives from the facts, as well as the various modes of reasoning. Being able to publish multi-dimensional data, such as statistics, on the web in a manner that allows for linking to related data sets and concepts is highly advantageous in numerous scenarios. The Data Cube vocabulary[3] offers a solution for achieving this using RDF. It serves as a framework that can be extended by additional vocabularies to facilitate the publication of other facets of (statistical) data flows or other multi-dimensional data sets.

Differently from previous work, in this paper, we propose a methodology for designing data platform models based on a preliminary version of an ontology we have developed. To the best of our knowledge, we are the first ones to propose a multi-level schema to support the dynamic creation of data platform models.

III. Dynamic Creation of Data Platforms Assets

The proposed methodology targets two primary objectives: i) describing the wide variety of assets that compose a modern data platform, and ii) enforcing structural relationships between these assets using a suitable ontology and supplementary constraints. Indeed, defining all asset relationships is paramount as they can be used to express asset composition or map a specific logical asset into many physical resources. The supporting ontology should include base assets (e.g., data collection, data product) and is easily extendable to include additional assets that might be needed in the future.

For instance, let's assume that we want to describe a new instance of a data product, such as *People*, describing the information about the personnel of a certain company. We first need to be able to define the relevant abstract type (e.g., *DataProductType*), which will be associated with a collection of relevant attributes (e.g., *domain*, *owner*). *People* will be assigned the *DataProductType* and may have, as values for those attributes, respectively *HumanResources* and *HumanResourceTeam*. We then also need to declare that *People* will serve as data product. Therefore, we need to be able to formalise the behaviour of a *DataProduct* asset and define the relations that this will have with other assets. For example, it may be linked to components such as its output ports (e.g., the tables *Persons* and *Salary*). For optimal flexibility, we thus need both the ability to define i) arbitrary types, ii) their function in the data platform, and iii) a set of relevant instances.

To address these objectives, we formulated an ontology structured into four different layers (*L0*, *L1*, *L2*, *L3*). The first layer of the ontology (*L0*) provides a mechanism for creating user-defined types. Those types will be defined in *L2*, which is used as a container for all possible data types. Each user-defined type will be associated with a name and a schema, where a schema is a list of pair names/types. In our model, a type can also be a structured type like a list, a structure, or an option (optional attribute). Thus, our model supports an arbitrary level of nesting. In the Big Data world, describing

[1]https://www.w3.org/TR/prov-o/. Last visited on 14 July 2023.

[2]https://www.w3.org/TR/vocab-dcat-2/. Last visited on 14 July 2023.

[3]https://www.w3.org/TR/vocab-data-cube/. Last visited on 14 July 2023.

assets using a structured approach like YAML or JSON is common. Our schema definition is functionally equivalent to what could be described using JSON schema.

The second layer of the ontology (*L1*) describes all the traits corresponding to the asset kinds we want to describe. A trait can be used to represent the behavior of a user-defined type. Specifically, *L1* formalizes the traits and defines all the possible structural relationships among traits. These will be automatically inherited by all the instances implementing those traits. While *L0* is meant to be stable since it defines the primitive methods for creating user-defined types, *L1* is expected to be modified by data engineers as they might add new traits in response to specific needs. *L0* and *L1* can be, in turn, used by data engineers or automatic tools for creating a set of user-defined types (*L2*) as well as their instances (*L3*).

IV. The Ontology

In this section, we will provide more details about AGILE-DM, a preliminary ontology that we have defined to describe data assets.

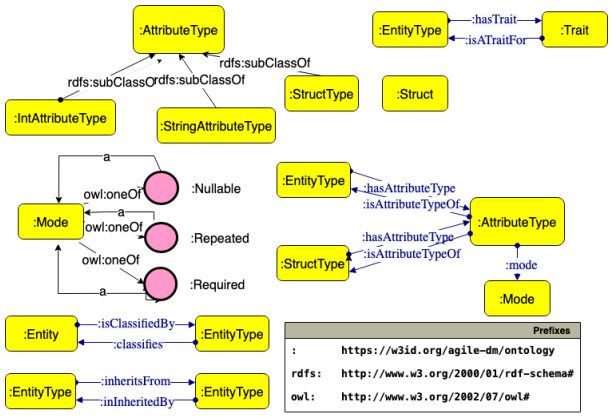

Fig. 1. AGILE-DM - L0.

As mentioned before, *L0* provide the basic mechanisms for dynamically creating new user-defined types, which in AGILE-DM are instances of `EntityType`. An `EntityType` instance can have different kinds of `AttributeType`.

Figure 1 shows two basic example of `AttributeType`: `IntAttributeType` and `StringAttributeType`. The first is used to encode integers, and the second to represent strings. `StructType` is an attribute type designed to model nested attributes.

The object property `hasAttributeType` has in the domain either an `EntityType` or a `StructType`. The range of `hasAttributeType` is represented by an `AttributeType`. This, in turn, has a data property `mode` that can be `Nullable`, `Required`, `Repeated`. *L0* also includes the following symmetrical relationships: `isClassifiedBy`, `inheritsFrom`, `hasAttributeType`, `hasTrait`. The `isClassifiedBy` relation is used to encode the type of a specific entity. The `inheritsFrom` relation denotes that one type is derived from another, following a single inheritance pattern. Operationally, the system merges the type schema with the inherited one. This enables to define common types and schemas for different assets.

The `hasTrait` relation allows us to define the specific traits of an `EntityType`. In practice, this is used for tagging an `EntityType` as a specific kind of asset in the data platform. Any possible asset that could populate a data platform is described as a trait. For example, we could treat a data lake as a flat list of data collections. Using our ontology, we could thus define a type `OurDataCollectionType` which will have the trait `DataCollection`. As a further example, a data platform based on the data mesh paradigm can be described as a list of data products where a data product contains a list of output ports, input ports, and workloads.

L1 contains all the possible traits, i.e., all the possible kinds of assets a data platform could manage. A trait can also model operational aspects of an asset like *versionable*, i.e., an asset needs to be versioned, and *deployable*, i.e., the asset needs to be deployed automatically as a whole.

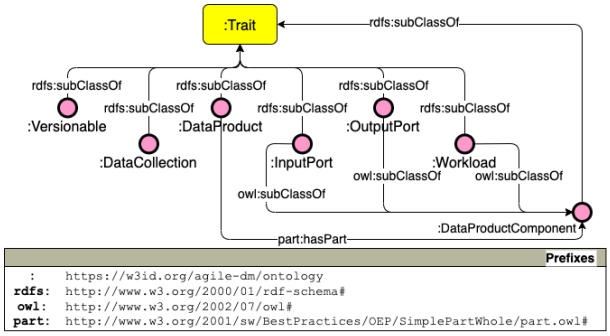

Fig. 2. AGILE-DM - L1.

Figure 2 shows seven traits representing data platform assets. For example, a data mesh is modeled as a collection of data products, where a data product can contain a list of components that can be either an `OutputPort`, and `InputPort`, or a `Workload`.

Therefore, AGILE-DM allows us both to define new types and to associate them with the trait that describes the correct asset. *L0* and *L1* can be used to create user-defined types (*L2*) and specific instances of them (*L3*). In the following, we present three examples of how *L2* and *L3* can be instantiated instantiated based on the abstract concepts in *L0* and *L1*.

Let us suppose we want to model a data platform asset `DataCollection`. In a broad sense, in a data platform, a `DataCollection` represents a simple container of data, either structured or unstructured. Following the data lake architectural style, a data lake is simply a collection of `DataCollection` instances. To keep these assets well-organized, it is important to properly describe a `DataCollection` in terms of its attributes. For instance, it is pretty common to organize a `DataCollection` in terms of ownership and domain, i.e., who is the owner of

the DataCollection and the domain the contained data belongs to. Let us suppose our metadata attributes are:

- name: data collection's name;
- organization: organization owning the data collection;
- domain: the domain the contained data belongs to.

We thus can defined DataCollectionType, an instance of EntityType with those attributes, as depicted in Figure 3.

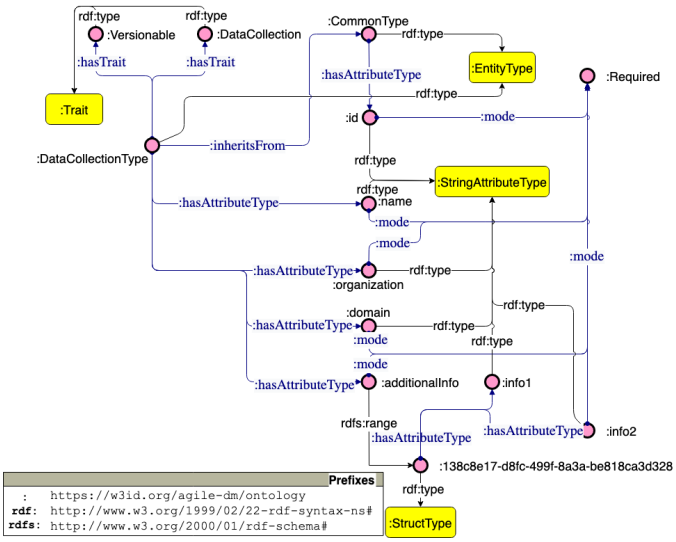

Fig. 3. AGILE-DM - L2. Example of DataCollectionType.

Next, we can create a specific instance of the user-defined type (DataCollectionType). Figure 4 shows two exemplary instances (at *L3*), produced on top of the three layers defined before. Each instance is associated with the specific attributes (e.g., name, domain, organization) specified by the DataCollectionType definition in Figure 3. This example also shows attribute nesting. In fact, the value of the property additionalInfo is an instance of the class Struct (defined in Figure 1), that can contain the attributes info1 and info2.

Finally, Figure 5 shows two exemplary instances of DataProductType and OutputPortType. Here we see a practical application of the traits defined in *L1*.

Specifically, in *L1* (see Figure 2), we defined traits corresponding to the data platform assets for the mesh architecture paradigm. In particular, we showed in *L1* that a DataProduct has a hasPart relationship with a trait DataProductComponent that could be three different things: OutputPort, InputPort or Workload. In turn, the DataProductType has a hasTrait relationship with the trait DataProduct and the OutputPortType has a hasTrait relationship with the trait OutputPort. Note that these traits were first defined in *L1* (see Figure 2).

V. Working Examples

In this section, we will show some functionalities of the platform that we are building on top of the AGILE-DM ontology to model all the typical assets composing a modern

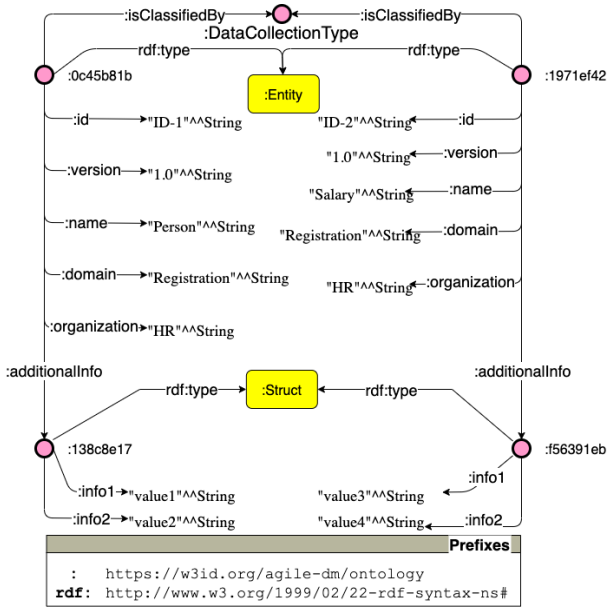

Fig. 4. AGILE-DM - L3. Examples of DataCollectionType entities.

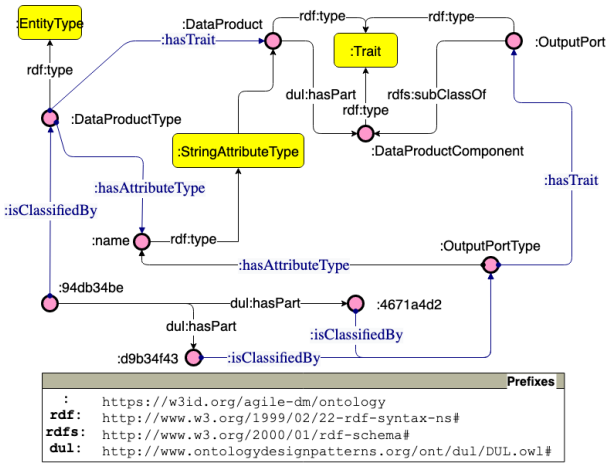

Fig. 5. AGILE-DM - L3. Examples of DataProductType and OutputPortType entities.

data platform. The aim is to showcase how the solution presented in this paper allows us to easily define a flexible and technology-independent description of assets, verify their consistency, and enable the generation of new instances.

The first example regards consistency verification. In our framework, a relationship between two instances of two types is legal if and only if the types have a hasTrait relationship with two Trait classes having that specific relationship between them (as shown in Figure 5). We can thus check that the generated elements of *L2* and *L3* are consistent by formulating a simple SPARQL query:

```
SELECT (COUNT(*) as ?count) WHERE {
 BIND(iri(instance1) as ?instanceId1)
 BIND(iri(instance2) as ?instanceId2)
  ?instanceId1 ns:isClassifiedBy ?type1 .
  ?instanceId2 ns:isClassifiedBy ?type2 .
  ?type1 ns:hasTrait ?trait1 .
```

```
  ?type2 ns:hasTrait ?trait2 .
  ?trait1 rdfs:subClassOf* ?tr1 .
  ?trait2 rdfs:subClassOf* ?tr2 .
  ?tr1 part:hasPart ?tr2 }
```

If it returns at least one element, it means that the `hasPart` relationship is legitimate among those two instances, i.e., their user-defined types are connected to two traits where that relationship has been defined. This mechanism is very flexible since the structural relationship among the assets can be defined at the trait level (*L1*).

Our platform also offers straightforward APIs for defining new types (at *L2* level) and generating their instances (at *L3* level). For example, for creating the `DataCollectionType` in Figure 3, we simply used the following Scala code:

```
service.create(
  EntityType(
    "CommonType", //Name of the type
    StructType(List( //The list of attributes
          "id" -> StringType() ) ) ) )
```

This creates the `CommonType`, which is going to be inherited by the `DataCollectionType` with the following code:

```
service.create(
  EntityType(
    "DataCollectionType",
    Set( //The list of Trait
      Versionable, DataCollection ),
    StructType( //Schema definition
      List(
        "name" -> StringType(),
        "organization" -> StringType(),
        "domain" -> StringType(),
        //A nested attribute
        "additional-info" -> StructType(
           List(
             "info1" -> StringType(),
             "info2" -> StringType()
        ) ) ) ), father)))
```

where `father` refers the `CommonType` previously defined.

Another API provides a generic way to link two instances with a relationship which is verified using the SPARQL query described above. The following code was used to create the entities in Figure 5:

```
//DataProductType definition
tms.create(
  EntityType(
    "DataProductType",
    Set(DataProduct),
    StructType(List("name" -> StringType())))))

//OutputPortType definition
tms.create(
  EntityType(
    "OutputPortType",
    Set(OutputPort),
    StructType(List("name" -> StringType())))))

//Creation of an instance of DataProductType
dp = ims.create("DataProductType",
                ("name" -> dataProduct1"))

//Creation of two instances of OutputPortType
op1 = ims.create(
        "OutputPortType", ("name" -> "outPort1"))
op2 = ims.create(
        "OutputPortType", ("name" -> "outPort2"))

//Link the DataProductType instance
//with the two OutputPortType instances
ims.link(dp, hasPart, op1) // consistency check
ims.link(dp, hasPart, op2) // consistency check
```

VI. Conclusions

In this paper, we introduced a novel methodology for characterizing a data platform using a multi-level formal ontology. Our proposed approach seeks to increase platform flexibility, rendering it technology-neutral and more receptive to technological shifts and advancements. We also outline the initial iteration of AGILE-DM, a new ontology developed for this objective. In future work, we plan to refine and publicly release the AGILE-DM ontology. Additionally, we aim to develop tools leveraging the ontology for dynamic instantiation of data platforms.

References

[1] K. Belhajjame, J. Cheney, D. Corsar, D. Garijo, S. Soiland-Reyes, S. Zednik, and J. Zhao, "Prov-o: The prov ontology," Tech. Rep., 2012. [Online]. Available: http://www.w3.org/TR/prov-o/

[2] B. Golshan, A. Halevy, G. Mihaila, and W.-C. Tan, "Data integration: After the teenage years," in *Proc. of the 36th ACM SIGMOD-SIGACT-SIGAI symposium on principles of database systems*, 2017, pp. 101–106.

[3] C. Peng, F. Xia, M. Naseriparsa, and F. Osborne, "Knowledge graphs: Opportunities and challenges," *AI Review*, pp. 1–32, 2023.

[4] D. Dessí, F. Osborne, D. R. Recupero, D. Buscaldi, and E. Motta, "Scicero: A deep learning and nlp approach for generating scientific knowledge graphs in the computer science domain," *Knowledge-Based Systems*, vol. 258, p. 109945, 2022.

[5] A. Kourid, S. Chikhi, and D. R. Recupero, "Fuzzy optimized v-detector algorithm on apache spark for class imbalance issue of intrusion detection in big data," *NCA*, vol. 35, no. 27, pp. 19 821–19 845, 2023. [Online]. Available: https://doi.org/10.1007/s00521-023-08783-8

[6] G. Xiao, D. Calvanese, R. Kontchakov, D. Lembo, A. Poggi, R. Rosati, and M. Zakharyaschev, "Ontology-based data access: A survey." in *IJCAI*, J. Lang, Ed. ijcai.org, 2018, pp. 5511–5519. [Online]. Available: http://dblp.uni-trier.de/db/conf/ijcai/ijcai2018.html#XiaoCKLPRZ18

[7] M. Lenzerini, "Data integration: A theoretical perspective." in *PODS*, L. Popa, S. Abiteboul, and P. G. Kolaitis, Eds. ACM, 2002, pp. 233–246. [Online]. Available: http://dblp.uni-trier.de/db/conf/pods/pods02.html#Lenzerini02

[8] L. Libkin, J. L. Reutter, A. Soto, and D. Vrgoc, "Trial: A navigational algebra for rdf triplestores." *ACM Trans. Database Syst.*, vol. 43, no. 1, pp. 5:1–5:46, 2018.

[9] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc, "Foundations of modern query languages for graph databases." *ACM Comput. Surv.*, vol. 50, no. 5, pp. 68:1–68:40, 2017.

[10] C. B. Aranda, "Federated query processing for the semantic web." Ph.D. dissertation, Madrid, Univ. Polite'cnica, 2014.

[11] C. Chen, B. Golshan, A. Y. Halevy, W.-C. Tan, and A. Doan, "Biggorilla: An open-source ecosystem for data preparation and integration." *IEEE Data Eng. Bull.*, vol. 41, no. 2, pp. 10–22, 2018.

[12] M. T. Özsu and P. Valduriez, "Distributed and parallel database systems." in *Computing Handbook, 3rd ed. (2)*, H. Topi and A. Tucker, Eds. CRC Press, 2014, vol. 13, pp. 1–24.

[13] S. Sakr, M. Wylot, R. Mutharaju, D. L. Phuoc, and I. Fundulaki, *Linked Data - Storing, Querying, and Reasoning*. Springer, 2018.

[14] K. Baclawski, E. S. Chan, D. Gawlick, A. Ghoneimy, K. Gross, Z. H. Liu, and X. Zhang, "Framework for ontology-driven decision making." *Appl. Ontology*, vol. 12, no. 3–4, pp. 245–273, 2017.

[15] J. Boyd, *Destruction and creation*. US Army Comand and General Staff College Leavenworth, WA, 1987.