Facade-X:
An Opinionated Approach to SPARQL Anything

Enrico Daga[1][0000−0002−3184−5407], Luigi Asprino[2][0000−0003−1907−0677],
Paul Mulholland[1][0000−0001−6598−0757], and Aldo Gangemi[3][0000−0001−5568−2684]

[1] The Open University (United Kingdom) {enrico.daga,paul.mulholland}@open.ac.uk
[2] University of Bologna (Italy) luigi.asprino@unibo.it
[3] Consiglio Nazionale delle Ricerche (CNR) aldo.gangemi@cnr.it

Abstract. The Semantic Web research community understood since its
beginning how crucial it is to equip practitioners with methods to trans-
form non-RDF resources into RDF. Proposals focus on either engineering
content transformations or accessing non-RDF resources with SPARQL.
Existing solutions require users to learn specific mapping languages (e.g.
RML), to know how to query and manipulate a variety of source for-
mats (e.g. XPATH, JSON-Path), or to combine multiple languages (e.g.
SPARQL Generate). In this paper, we explore an alternative solution
and contribute a general-purpose meta-model for converting non-RDF
resources into RDF: *Facade-X*. Our approach can be implemented by
overriding the SERVICE operator and does not require to extend the
SPARQL syntax. We compare our approach with the state of art meth-
ods RML and SPARQL Generate and show how our solution has lower
learning demands and cognitive complexity, and it is cheaper to imple-
ment and maintain, while having comparable extensibility and efficiency.

Keywords: SPARQL · Meta-model · Re-engineering

1 Introduction

Knowledge graphs have nowadays a key role in domains such as enterprise data
integration and cultural heritage. However, domain applications typically deal
with heterogeneous data objects. Therefore, ontology engineers develop knowl-
edge graph construction pipelines that include the transformation of different
types of content into RDF. Typically, this is achieved by using tools that act as
mediators between the data sources and the needed format and data model [12].
Alternatively, dedicated software components implement ad-hoc transformations
from custom formats to a multiplicity of ontologies relevant to the domain [4].
We place our research under the context of the EU H2020 SPICE project, which
aims at developing a linked data infrastructure for integrating and leveraging
museum collections using multiple ontologies covering sophisticated aspects of
citizen engagement initiatives [4]. Museum collections come in a variety of data
objects, spanning from public websites to open data sets. These include metadata

---

[4] SPICE Project: https://spice-h2020.eu

summaries as CSVs, record details as JSON files, and binary objects (e.g. artwork images), among others. The semantic lifting of such a variety of resources can be a serious bottleneck for the project activities. Several languages have been developed to either engineer content transformation (e.g. RML) or extending the SPARQL query language to access non-RDF resources (e.g. SPARQL Generate). However, existing solutions require Semantic Web practitioners to learn a mapping language, or even combine multiple languages, for example requiring to use XPath for XML transformations. In addition, these require Semantic Web practitioners to know the details of the original format (e.g. XML) as well as the target domain ontology.

In this paper, we don't propose a new language. Instead, we aim at reducing the effort of Semantic Web practitioners in dealing with heterogeneous data sources by providing a generic, domain-independent meta-model as a *facade* to wrap the original resource and to make it query-able *as-if* it was RDF. Specifically, we contribute a meta-model and associated algorithm for accessing non-RDF resources as RDF: *Facade-X*. Our approach can be implemented by overriding the SERVICE operator and does not require to extend the SPARQL syntax. We compare our approach with the state of art methods RML and SPARQL Generate, and show how our solution has lower learning demands and cognitive complexity, and it is cheaper to implement and maintain, while having comparable extensibility and efficiency (in our naive implementation).

In the next section we analyse the key requirements, building also on the work of [15]. In Section 3 we describe our approach for adopting *facades* for re-engineering resources into RDF and give a formal definition of *Facade-X*. Section 4 is dedicated to the prototype implementation of the approach in a software named SPARQL Anything. Related work is discussed in Section 5. We compare our approach with state of art methods (RML and SPARQL Generate) in Section 6, before concluding our paper in Section 7.

2   Requirements

The motivation for researching novel ways to transform non-RDF resources into RDF comes from the scenarios under development in the EU H2020 project SPICE: Social Cohesion, Participation, and Inclusion for Cultural Engagement. In this project, a consortium of eleven partners collaborate in developing novel ways for engaging with cultural heritage, relying on a *linked data* network of resources from museums, social media, and businesses active in the cultural industry. However, the majority of resources involved are not exposed as Linked Data but are released, for example, as CSV, XML, JSON files, or combinations of these formats. In addition, the research activity aims at the design of task-oriented ontologies, producing multiple semantic *viewpoints* on the resources and their metadata. It is clear how the effort required for transforming resources could constitute a significant cost to the project. In the absence of a strategy to cope with this diversity, content transformation may result in duplication of effort and become a serious bottleneck. Table 1 provides a summary of the requirements. The main requirement is the ability to support users in transforming existing

Table 1: Requirements

Requirement	Description
`Transform`	Transform several sources having heterogeneous formats
`Query`	Query resources having heterogeneous formats
`Binary`	Support the transformation of binary formats
`Embed`	Support the embedding of content in RDF
`Metadata`	Support the extraction of metadata embedded in files
`Low learning demands`	Minimise the tools and languages that need to be learned
`Low complexity`	Minimise complexity of the queries
`Meaningful abstraction`	Enable focus on data structures rather than implementation details
`Explorability`	Enable data exploration without premature commitment to a mapping, in the absence of a domain ontology.
`Workflow`	Integrate with a typical Semantic Web engineering workflow
`Adaptable`	Be generic but flexible and adaptable
`Sustainable`	Inform into a software that is easy to implement, maintain, and does not have evident efficiency drawbacks
`Extendable`	Support the addition of an open set of formats

non-RDF resources having heterogeneous formats (`Transform`). In addition, the solution should be able to support cases in which practitioners only need to interrogate the content (`Query`). A valid approach should be able to cope with binary resources as well as textual formats (`Binary`). In the cultural heritage domain, metadata files are typically associated to repositories of binary content such as images in various formats. Applications may need to transfer data and metadata in a single operation, embedding the binary content in a data value (`Embed`) and extracting metadata (`Metadata`) from the file (from EXIF annotations).

We consider requirements related to usability and adoption. The approach should ideally limit the number of new languages and tools that need to be learned in order to transform and use non-RDF resources (`Low learning demands`). This can be expected to both encourage adoption and reduce the learning curve for new users. The code that the user is required to develop in order to access the resources should be as simple as possible (`Low complexity`). The approach should provide the user with a meaningful level of abstraction, enabling them to focus on the the structure of the data (e.g. data rows and hierarchies) rather than the details of how the structure has been implemented (`Meaningful abstraction`). The approach should support an exploratory way of working in which the user does not have to prematurely commit to a domain ontology before they come to understand the data representation that they require (`Explorability`). The resulting technology should be easily combined with typical Semantic Web engineering workflow (`Workflow`). This requirement, already mentioned in [15], is interpreted considering that the solution should rely as much as possible on already existing technologies typically used by our domain users. The approach should allow for a technical solution that is generic but

easily `Adaptable` to user tasks, for example, supporting symbol manipulation, variable assignments, and data type manipulation.

Finally, we look into requirements of software engineering. The approach should be `Sustainable` and inform a software that is easy to implement on top of existing Semantic Web technologies, easy to maintain, and does not have efficiency drawbacks compared to alternative state of the art solutions. Ultimately, the system should be easy to extend (`Extendable`) to support an open ended set of formats.

3   An Opinionated Approach

We introduce a novel approach to interrogate non-RDF resources with SPARQL. Our opinion is that the task of transforming resources into RDF should be decoupled in two very different operations: (a) re-engineering, and (b) remodelling. We define *re-engineering* as the task of transforming resources minimising domain considerations, focusing on the meta-model. Instead, remodelling is the transformation of *domain knowledge*, where the original domain model is reframed into a new one, whose main objective is to add semantics. From this perspective, we propose to solve the re-engineering problem automatically and delegating the remodelling to the RDF-aware user. *How to use RDF to access heterogeneous source formats?* We rely on the notion of *facade* [14] as *"an object that serves as a front-facing interface masking more complex underlying or structural code"*[5]. Applied to our problem, a facade acts as a generic meta-model allowing (a) to inform the development of transformers from an open ended set of formats, and (b) to generate RDF content in a consistent and predictable way. In what follows, we describe a generic approach that can be used to develop facade-based connectors to heterogeneous file formats. After that, we introduce Facade-X, which is the first of these interfaces, and describe how our facade maps to RDF. Finally, we design a method to inject facades into SPARQL engines. To support the reader, we introduce a guide scenario reusing the data of the Tate Gallery collection, published on GitHub[6]. The repository contains CSV tables with metadata of artworks and artists and a set of JSON files with details about each catalogue record, for example, with the hierarchy of archive subjects. The file `artwork_data.csv` includes metadata of the artworks in the collection such as `id`, `artist`, `artistId`, `title`, `year`, `medium`, and references two external resources: a JSON file with the artwork `subjects` headings and a link to a JPG `thumbnail` image. Our objective is to serve this content to the Semantic Web practitioners for exploration and reuse.

3.1   Resources, data sources, and facades

In this section we give a formal definition of the three components of our approach: resources, data sources, and facades. In addition, we describe how an algorithm can apply these concepts for re-engineering resources in RDF.

---

[5] See also The Facade Design Pattern: https://en.wikipedia.org/wiki/Facade_pattern (accessed 15/12/2020).

[6] Tate Gallery collection metadata: https://github.com/tategallery/collection.

We consider a *resource* anything accessible from a URL and distinguish it from its content, that we name *data source*. The file `artwork_data.csv`[7] and the image $N04858\_8.jpg$ are resources and the CSV and JPG content are data sources. We assume that a resource contains at least one data source[8]. A data source can be named with the URL or have a different name[9]. We introduce the following predicates and associated axioms, in predicate logic:

```
1   Resource(r)  DataSource(ds)  Name(n)  includes(r, ds)  hasName(ds, n)
2   ∀r.Resource(r) → ∃ds.includes(r, ds) ∧ DataSource(ds)
3   ∀ds.DataSource(ds) → ∃n.hasName(ds, n) ∧ Name(n)
4   ∀ds.DataSource(ds) → ∃r.Resource(r) ∧ includes(r, ds)
5   ∀ds₁∀ds₂∀n∀r.
6     includes(r, ds₁) ∧ includes(r, ds₂) ∧ hasName(ds₁, n) ∧ hasName(ds₂, n) → ds₁ = ds₂
7   ∀ds∀n₁∀n₂.hasName(ds, n₁) ∧ hasName(ds, n₂) → n₁ = n₂
```

In addition, we refer to two additional concepts: RDF Graph and RDF Dataset, as specified by RDF 1.1 [3]. We now describe how an algorithm can apply facades to resources to derive RDF datasets capable of answering a given query[10]. Let Q be the set of all possible queries, G the set of all possible graphs, N the set of all possible graph names, R the set of all possible resources and DS the set of all data sources (found in the resources). We define: *(i)* D as a collection of named graphs (i.e. $D \subseteq N \times G$); *(ii)* A (i.e. the algorithm) as a function that given a resource ($r \in R$), a facade function ($f \in F$), and a query ($q \in Q$), returns a collection of named graphs including the graphs required to answer the query (i.e. one of the possible subsets of $N \times G$); *(iii)* F is a set of functions where each $f \in F$ associates a data source from the resource ($ds \in DS$) and a query ($q \in Q$) with a graph $g \in G$, according to a facade. A and F can be formally defined as follows:

$$A : R \times F \times Q \to 2^{N \times G} \qquad\qquad F = \{f | f : DS \times Q \to G\}$$

Additionally, given a query $q \in Q$, a resource $r \in R$ and its data sources $ds \in DS$, we define: *(i)* $g^*_{ds,q} \in G$ as the graph which contains the minimal (optimal) set of triples required to answer q on ds; *(ii)* $D^*_{r,q} = \{(n, g^*_{ds,q}) | includes(r, ds)$ and $n \in N$ and $g^*_{ds,q} \in G\}$ as the collection of minimal set of triples required to answer q on r. It is worth noticing that given a query and a resource neither A nor any $f \in F$ has to return an optimal response (i.e. $D^*_{r,q}$ and $g^*_{ds,q}$), but they can return any super set of the optimum (i.e. any $g \in G$ such that $g^*_{ds,q} \subseteq g$). We don't make any commitment on the underlying implementation of the facade with respect to the resource/data sources, apart from assuming that the resulting dataset will be sufficient, but not necessarily optimal, for answering the query.

---

[7] Available at https://raw.githubusercontent.com/tategallery/collection/master/artwork_data.csv

[8] In principle, a resource may include multiple data sources, for example, an Excel spreadsheet may include several sheets.

[9] Although resources and data sources can be named with the same string (URL), we consider them different entities in our model.

[10] Note that we are not enforcing a specific algorithm, although we implement one in our experimental evaluation.

3.2   Facade-X

We base the design of Facade-X on the distinction between containers and values. Specifically, we define a *container* as a set of uniquely identifiable *slots*, each one of them including either another container or a data *value*. Slot identifiers (*keys*) can be either XSD strings (*StringKey*) or XSD positive integers (*NumberKey*). The predicate *Key* is a reification of either an integer or a string, while the predicate *Value* reifies a string only. Containers can optionally be qualified by a *type*. In Facade-X, data sources are referred to as *root* containers. We specify our facade in predicate logic as follows:

```
1   Root(c0)  Container(c1)  Slot(s1)  Key(n)
2   StringKey(n)  NumberKey(n)  Value(v1)  Type(t)
3   ∀k.StringKey(k) → Key(k)
4   ∀k.NumberKey(k) → Key(k)
5   ¬∃k.NumberKey(k) ∧ StringKey(k)
6   ∀ c.Root(c) → Container(c)
```

In addition, we define relations between the model components, including definitions of domain and range:

```
1   hasSlot(c, s)  hasType(c, t)  hasKey(s, k)
2   hasContainer(s, c)  hasValue(s, v)
3   ∀(x, y).hasSlot(x, y) → Container(x) ∧ Slot(y)
4   ∀(x, y).hasType(x, y) → Container(x) ∧ Type(y)
5   ∀(x, y).hasKey(x, y) → Slot(x) ∧ Key(y)
6   ∀(x, y).hasContainer(x, y) → Slot(x) ∧ Container(y)
7   ∀(x, y).hasValue(x, y) → Slot(x) ∧ Value(y)
```

We define a set of axioms describing additional properties of the meta-model. Only containers can have a type (but they don't have to), and there can only be one root container. A slot can have either one container or one value and cannot have both. A slot can be member of one container only and slots of a container are uniquely identified by their key:

```
1   ∀(x, y).Root(x) ∧ Root(y) → x = y
2   ¬∃(x, y, z).hasContainer(x, y) ∧ hasValue(x, z)
3   ∀(x, y, z).hasContainer(x, y) ∧ hasContainer(x, z) → y = z
4   ∀(x, y, z).hasValue(x, y) ∧ hasValue(x, z) → y = z
5   ∀(x, y, z).hasSlot(x, y) ∧ hasSlot(z, y) → x = z
6   ∀(c, s1, s2p, n).hasSlot(c, s1) ∧ hasSlot(c, s2) ∧ hasKey(s1, n) ∧ hasKey(s2, n) → s1 = s2
```

The data from our guide scenario can be represented as follows:

```
1   Root(ds)
2   StringKey(id)  StringKey(artist)  StringKey(artistId)  StringKey(title)
3   hasSlot(ds, s1)  hasKey(s1, IntegerKey(1))  hasContainer(s1, r1)
4        hasSlot(r1, r1s1)  hasKey(r1s1, id)  hasValue(r1s1, "1035")
5        hasSlot(r1, r1s2)  hasKey(r1s2, artist)  hasValue(r1s2, "Blake Robert")
6        hasSlot(r1, r1s3)  hasKey(r1s3, artistId)  hasValue(r1s3, "38")
7        hasSlot(r1, r1s4)  hasKey(r1s4, title)  hasValue(r1s4, "A Figure Bowing ...")  [...]
```

Finally, we define mapping rules to RDF, where properties are built using string keys and resources can be either blank nodes or named IRIs[11]:

---

[11] The system may allow users to define their own namespace, or reuse the name of the *ds*, and leave to the underlying machinery to mint IRIs.

$$1 \quad Root(ds) \xrightarrow{f} Triple(Resource(ds), rdf:type, fx:Root)$$
$$2 \quad hasSlot(c,s) \wedge hasKey(s,k) \wedge StringKey(k) \wedge hasContainer(s,c1)$$
$$3 \quad \quad \xrightarrow{f} Triple(Resource(c), Property(k), Resource(c1))$$
$$4 \quad hasSlot(c,s) \wedge hasKey(s,k) \wedge IntegerKey(k) \wedge hasContainer(s,c1)$$
$$5 \quad \quad \xrightarrow{f} Triple(Resource(c), ContainerMembershipProperty(k), Resource(c1))$$
$$6 \quad hasSlot(c,s) \wedge hasKey(s,k) \wedge StringKey(k) \wedge hasValue(s,v)$$
$$7 \quad \quad \xrightarrow{f} Triple(Resource(c), Property(k), Literal(v))$$
$$8 \quad hasSlot(c,s) \wedge hasKey(s,k) \wedge IntegerKey(k) \wedge hasValue(s,v)$$
$$9 \quad \quad \xrightarrow{f} Triple(Resource(c), ContainerMembershipProperty(k), Literal(v))$$
$$10 \quad hasType(c,t) \xrightarrow{f} Triple(Resource(c), rdf:type, Resource(t))$$

Our model maps into an RDF that mixes *lists*, *type* statements, and *key-value* pairs. Recent work suggests good practices for developing lists in RDF that are efficient to query [5, 6], favouring container membership properties over nested structures to represent lists. We define two namespaces, one for the primitive entity `Root` and another for minting properties from keys[12]. The Facade-X RDF vocabulary is published at http://sparql.xyz/facade-x/ns/. The above mappings produce the following *Facade-X RDF*, from our example scenario:

```
1  @prefix fx: <http://sparql.xyz/facade-x/ns/>.
2  @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
3  @base <http://sparql.xyz/facade-x/data/>.
4  [] a fx:Root ;
5     rdf:_1 [:id "1034"; :artist "Blake Robert"; :artistId "38"; ...
6     rdf:_2 [:id "16216"; :artist "Williams Terrick" :artistId "2149"; ...
7     rdf:_3 [:id "12086"; :artist "Pissarro Lucien" :artistId "1777"; ...
8     ...
```

3.3 Using facades in SPARQL

The algorithm in Section 3.1 requires as input a URL and returns an RDF dataset as output. We propose to *overload* the SPARQL SERVICE operator by defining a custom URI-schema, based on the protocol `x-sparql-anything:`, which is intended to behave as a *virtual* remote endpoint. The related URI-schema supports an open-ended set of parameters specified by the facade implementations available. Options are embedded as key-value pairs, separated by comma. Implementations are expected to either guess the source type from the resource locator or to obtain an indication of the type from the URI schema, for example, with an option "mime-type":

```
x-sparql-anything:mime-type=application/json; charset=UTF-8,location=...
```

Following our example scenario, users can write a query and select metadata from the CSV file, as well as embed the content of remote JPG thumbnails in the RDF. Multiple SERVICE clauses may integrate data from more files, for example, the JSON with details about artwork subjects. We leave the content of the CONSTRUCT section to be filled by the ontology engineer:

---

[12] Not all strings are valid IRI local names. Implementations will need to apply heuristics to cope with corner cases in CSV or JSON keys.

```
1   PREFIX fx: <http://sparql.xyz/facade-x/ns/>
2   PREFIX xyz: <http://sparql.xyz/facade-x/data/>
3   PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
4   CONSTRUCT {
5       [...] # Amazing ontology here
6   } WHERE {
7       BIND (IRI(CONCAT(STR(tate:), "artwork-", ?id )) AS ?artwork) .
8       BIND (IRI(CONCAT(STR(tate:), "artist-", ?artistId )) AS ?artist) .
9       SERVICE <x-sparql-anything:csv.headers=true,location=file:./
            artwork_data.csv> {
10          []   xyz:id ?id ;              xyz:artist ?artistLabel ;
11               xyz:accessionId ?accId ;  xyz:artistId ?artistId ;
12               xyz:title ?title;         xyz:medium ?medium ;
13               xyz:year ?year ;          xyz:thumbnailUrl ?thumbnail .
14      }
15      # JPEG Thumbnail from the Web
16      BIND (IRI(CONCAT("x-sparql-anything:location=", ?thumbnail )) AS ?
            embedJPG ).
17      SERVICE ?embedJPG { [] rdf:_1 ?imageInBase64 }.
18      # JSON File with subjects
19      BIND (IRI(CONCAT("x-sparql-anything:file:./artworks/", ?accId )) AS ?
            subJSON ).
20      SERVICE ?subJSON { [ xyz:id ?subjectId ; xyz:name ?subjectName ] }.
21  }
```

4   Implementation to SPARQL Anything

In this section we describe SPARQL Anything which is meant to provide a
proof-of-concept of our approach. SPARQL Anything implements a stack of
transformers mapped to media types and file extensions. The framework al-
lows the addition of an open-ended set of transformers as Java classes. Dur-
ing execution, a query manager intercepts usage of the SERVICE operator
and in case the endpoint URI has the `x-sparql-anything` protocol, it parses
the URI extracting the resource locator and parameters. Default parameters
are: `mime-type`, `locator`, `namespace` (to be used when defining RDF re-
sources), and `root` (to use as the IRI of the root RDF resource, instead of a
blank node), and `metadata`. SPARQL Anything will project an RDF dataset
during query execution including the data content and optionally a graph
named `http://sparql.xyz/facade-x/data/metadata`, including file metadata
extracted from image files (also in Facade-X). Specific formats may support spe-
cific parameters. For example, the Text triplifier supports a regular expression
to be used by a tokenizer that splits the content in a list of strings (defaults to
the space character). Similarly, the CSV triplifier allows to specify whether to
use the first row as headers or only use column indexes. More information on
the currently supported formats can be found in the project page[13].

   We validated the generality of Facade-X as a meta-model with relation to the
triplifiers currently implemented in SPARQL Anything. We already considered
CSV in the guide example. The following JSON example, also derived from the
Tate Gallery open data, can be mapped to our model as in the associated listing.

```
1   { "fc": "Kazimir Malevich",
2      "id": 1561,
```

---

[13] http://github.org/sparql-anything/sparql-anything.

```
3     "places": [
4       { "name": "Ukrayina", "type": "nation" },
5       { "name": "Moskva, Rossiya", "type": "inhabited_place" }
6     ]}
7
```
8 $Root(malevic)$
9 $StringKey(fc) StringKey(id) StringKey(places) StringKey(name) StringKey(type)$
10 $NumberKey(1) NumberKey(2)$
11 $hasSlot(malevic, s_{fc}) \wedge hasKey(s_{fc}, fc) \wedge hasValue(s_{fc}, "Kazimir Malevich")$
12 $hasSlot(malevic, s_{id}) \wedge hasKey(s_{id}, id) \wedge hasValue(s_{id}, 1561)$
13 $hasSlot(malevic, s_{places}) \wedge hasKey(s_{places}, places) \wedge hasContainer(s_{places}, c_{places})$
14 $hasSlot(c_{places}, s_{place/1}) \wedge hasKey(s_{place/1}, 1) \wedge hasContainer(s_{place/1}, ukraina)$
15 $hasSlot(ukraina, s_{ukr/name}) \wedge hasKey(s_{ukr/name}, name) \wedge$
16 $\qquad hasValue(s_{ukr/name}, "Ukrayina")$
17 $hasSlot(ukraina, s_{ukr/type}) \wedge hasKey(s_{ukr/type}, type) \wedge hasValue(s_{ukr/type}, "nation")$
18 $hasSlot(c_{places}, s_{place/2}) \wedge hasKey(splace/2, 2) \wedge hasContainer(s_{place/2}, moskva)$
19 $hasSlot(moskva, s_{mos/name}) \wedge hasKey(s_{mos/name}, name) \wedge$
20 $\qquad hasValue(s_{mos/name}, "Moskva, Rossiya")$
21 $hasSlot(moskva, s_{mos/type}) \wedge hasKey(s_{mos/type}, type) \wedge$
22 $\qquad hasValue(s_{mos/type}, "inhabited\_place")$

Finally, we show how an excerpt from a catalogue record in XML, can be interpreted with our Facade-X (this also applies to HTML):

```
1     <OGT hint="OGGETTO">
2       <OGTD hint="Definizione">reperti antropologici ...</OGTD>
3       <OGTT hint="Tipologia">reperto osteo-dentario</OGTT>
4       ...
5
```
6 $Root(ogt) \wedge hasType(record, "OGT")$
7 $Container(ogtd) \wedge hasType(ogtd, "OGTD")$
8 $Container(ogtt) \wedge hasType(ogtt, "OGTT")$
9 $StringKey(hint)$
10 $NumberKey(1) NumberKey(2)$
11 $hasSlot(ogt, s_1) \wedge hasKey(s_1, 1) \wedge hasValue(s_1, ogtd)$
12 $hasSlot(ogt, s_2) \wedge hasKey(s_2, 2) \wedge hasValue(s_2, ogtt)$
13 $hasSlot(ogt, s_{hint}) \wedge hasKey(s_{hint}, hint) \wedge hasValue(s_{hint}, "OGGETTO")$
14 ...

5 Related Work

Related work includes semantic web approaches to content re-engineering, approaches to extending the functionalities of SPARQL, and research on end-user development and human interaction with data.

In ontology engineering, non-ontological resource re-engineering refers to the process of taking an existing resource and transforming it into an ontology [21]. These family of approaches integrate resource transformation within the methodology, where domain knowledge plays a central role. Triplify [2] is one of the first tools aiming at converting sources into RDF in a domain independent way. The approach is based on mapping HTTP URIs to ad-hoc database queries, and rewriting the output of the SQL query into RDF. Other tools are based on the W3C Direct Mapping recommendation [20] for relational databases. Systems are available for automatically transforming data sources of several formats into RDF (Any23[14], JSON2RDF[15], CSV2RDF[16] to name a few). A recent survey

---

[14] http://any23.apache.org/

[15] https://github.com/AtomGraph/JSON2RDF

[16] http://clarkparsia.github.io/csv2rdf/

lists systems to lift tabular data [9]. While these tools have a similar goal (i.e. enabling the user to access the content of a data source as if it was in RDF), the (meta)model used for generating the RDF data highly depends on the input format. All these approaches are not interested in the requirement of providing a common useful abstraction to heterogeneous formats. A long history of mapping languages for transforming heterogeneous files into RDF can be considered superseded by RML [8], including a number of approaches for ETL-based transformations [1]. We consider RML as representative of general data integration approaches such as OBDA [24]. This family of solutions are based on a set of declarative mappings. The mapping languages incorporate format-specific query languages (e.g. SQL or XPath) and require the practitioner to have deep knowledge not only of the input data model but also of standard methods used for its processing. Recent work acknowledges how these languages are built with machine-processability in mind [13] and how defining or even understanding the rules is not trivial to users.

We survey approaches to extend SPARQL. A standard method for extending SPARQL is by providing custom functions[17], or by using so-called *magic* properties. This approach defines custom predicates to be used for instructing specific behaviour at query execution. SPARQL-Generate [15] introduces a novel approach for performing data transformation from heterogeneous sources into RDF by extending the SPARQL syntax with new operators [15]: GENERATE, SOURCE, and ITERATOR. Custom functions perform ad-hoc operations on the supported formats, for example, relying on XPath or JSONPath. Other approaches extend SPARQL *without* changes to the standard syntax. BASIL [7] allows to define parametric queries by enforcing a convention in SPARQL variable names. As a result, SPARQL query templates can be processed with standard query parsers. `grlc` decorates the query with execution metadata declared in comments [17]. SPARQL Micro-service [18] provides a framework that, on the basis of API mapping specification, wraps web APIs in SPARQL endpoints and uses JSON-LD profile to translate the JSON responses of the API into RDF. In this paper, we follow a similar, minimalist approach and extend SPARQL by *overriding* the behaviour of the SERVICE operator. We compare our proposal with SPARQL Generate and RML in detail in the evaluation section.

The proposed approach relates to research on end-user development and human interaction with data. End-user development is defined by [16] as *"methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify or extend a software artefact"*. Many SPARQL users fall into the category of end-user developer. In a survey of SPARQL users, [22] found that although 58% came from the computer science and IT domain, other SPARQL users came from non-IT areas, including social sciences and the humanities. Findings in this area [19] suggest

---

[17] ARQ provides a library of custom functions for supporting aggregates such as computing a standard deviation of a collection of values. ARQ functions: https://jena.apache.org/documentation/query/extension.html (accessed 15/12/2020).

that the data with which users work is more often primarily list-based and/or hierarchical rather than tabular. For example, [11] proposes an alternative formulation to spreadsheets in which data is represented as *list-of-lists*, rather than tables. Our proposal goes in this direction and accounts for recent findings in end-user development research.

6   Evaluation

We conduct a comparative evaluation of SPARQL Anything with respect to the state of art methods RML and SPARQL Generate. First, we analyse in a quantitative way the cognitive complexity of the frameworks. Second, we conduct a performance analysis of the reference implementations. Finally, we discuss the approaches in relation to the requirements elicited in Section 2. Competency questions, queries, experimental data, and code used for the experiment are available on the GitHub repository of the SPARQL Anything project[18].

Cognitive Complexity Comparison. We present a quantitative analysis on the cognitive complexity of SPARQL Anything, SPARQL Generate and RML frameworks. One effective measure of complexity is the number of distinct items or variables that need to be combined within a query or expression [10]. Such a measure of complexity has previously been used to explain difficulties in the comprehensibility of Description Logic statements [23]. Specifically, we counted the number of tokens needed for expressing a set of competency questions. We selected four JSON files from the case studies of the SPICE project where each file contains the metadata of artworks of a collection. Each file is organised as a JSON array containing a list of JSON objects (one for each artwork). This simple data structure avoids favouring one approach over the others. Then, an analysis of the schema of the selected resources allowed us to define a set of 12 competency questions (CQs) that were then specified as SPARQL queries or mapping rules according to the language of each framework, in particular: (i) 8 CQs (named q1-q8), aimed at retrieving data from the sources, were specified as SELECT queries (according to SPARQL Anything and SPARQL Generate); (ii) 4 CQs (named q9-q11), meant for transforming the source data to RDF, were expressed as CONSTRUCT queries (according to SPARQL Anything and SPARQL Generate) or as mapping rules complying with RML. These queries/rules intend to generate a blank node for each artwork and to attach the artwork's metadata as dataproperties of the node. Finally, we tokenized the queries (by using "(){},;\n\t\r␣ as token delimiters) and we computed the total number of tokens and the number of distinct tokens needed for each queries. By observing the average number of tokens per query we can conclude that RML is very verbose (109.75 tokens) with respect to SPARQL Anything (26.25 tokens) and SPARQL Generate (30.75 tokens) whose verbosity is similar (they differ of the ~6.5%). However, the average number of *distinct* tokens per query shows that SPARQL Anything requires less cognitive load than other frameworks. In

---

[18] https://github.com/spice-h2020/SPARQL Anything/tree/main/experiment

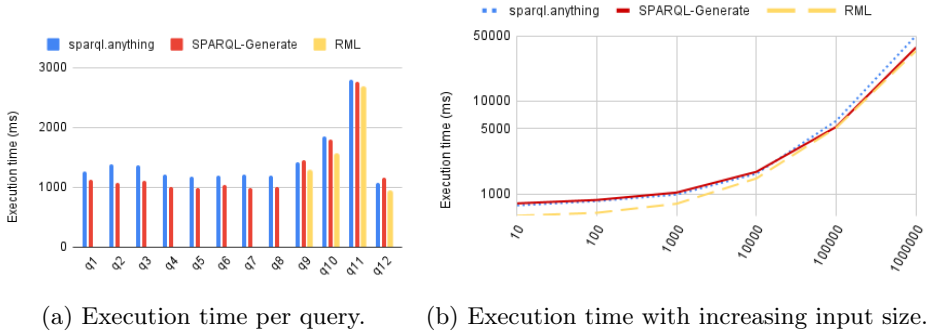(a) Execution time per query. (b) Execution time with increasing input size.

Fig. 1: Analysis of the the execution time.

Table 2: Formats supported by RML, SPARQL Generate, and SPARQL Anything.

	JSON	CSV	HTML	Bin.	XML	RDB	Text	Embed	Meta.	Spread.
RML	✓	✓	✓		✓	✓	✓			
SPARQL-Generate	✓	✓	✓	✓	✓	✓	✓			
SPARQL Anything	✓	✓	✓	✓	✓		✓	✓	✓	✓

fact, while SPARQL Anything required 18.25 distinct tokens, SPARQL Generate needed 25.5 distinct tokens (~39.72% more) and RML 45.25 distinct tokens (~150% more).

Performance Comparison. We assessed the performance of three frameworks in generating RDF data. All of the tests described below were run three times and the average time among the three executions is reported. The tests were executed on a MacBook Pro 2020 (CPU: i7 2.3 GHz, RAM: 32GB). Figure 1a shows the time needed for evaluating the SELECT queries q1-q8 and for generating the RDF triples according to the CONSTRUCT queries/mapping rules q9-q12. The three frameworks have comparable performance. We also measured the performance in transforming input of increasing size. To do so, we repeatedly concatenated the data sources in order to obtain a JSON array containing 1M JSON objects and we cut this array at length 10, 100, 1K, 10K and 100K. We ran the query/mapping q12 on these files and we measured the execution time shown in Figure 1b. We observe that for inputs with size smaller than 100K the three frameworks have equivalent performance. With larger inputs, SPARQL Anything is slightly slower than the others. The reason is that, in our naive implementation, the data source is completely transformed and loaded into a RDF dataset in-memory, before the query is evaluated. However, implementations could *stream* the triples during query execution, or transform the optimal triple set for the query solution, thus achieving better performance on large input. However, we leave this optimisations to future work.

Requirements satisfaction and discussion We discuss the requirements introduced in Section 2.

`Transform`, `Binary`, `Embed`, and `Metadata`. All the frameworks support users in transforming heterogeneous formats with few differences (a comparison is provided in Table 2). Currently, SPARQL Anything and SPARQL-Generate cover the largest set of input formats. SPARQL-Generate however does not support embedding content (`Embed`) and extracting metadata from files (`Metadata`). Both features are not supported by RML, which doesn't support plain text as well. SPARQL Anything allows users to query spreadsheets, but it is not able to handle relational databases yet[19]. SPARQL Anything is the only tool supporting the extraction of metadata and the embedding of binary content.

`Query`. In terms of query support, while RML requires data to be transformed first and then uploaded to a SPARQL triple store, SPARQL Anything and SPARQL-Generate enable users to query resources directly.

`Low learning demands`. SPARQL Generate uses an extension to SPARQL 1.1 to transform source formats into RDF. RML provides an extension to the R2RML vocabulary in order to map source formats into RDF. Therefore either a SPARQL extension or a new mapping language has to be learned to perform the translation. In the case of Facade-X, no new language has to be learned as data can be queried using existing SPARQL 1.1 constructs.

`Low complexity`. Complexity can be measured as the number of distinct items or variables that need to be combined with the query. In experiments, Facade-X is found to perform favourably in comparison to SPARQL Generate and RML.

`Meaningful abstraction`. Differently from RML and SPARQL-Generate, which require users to be knowledgeable of the source formats and their query languages (e.g. XPath, JSONPath etc.), Facade-X users can access a resource *as if* it was an RDF dataset, hence the complexity of the non-RDF languages is completely hidden to them. The cost for this solution is limited to the users which are required to *explore* the facade that is generated and tweak the configuration via the Facade-X IRI schema.

`Explorability`. With SPARQL Generate and RML, the user needs to commit to a particular mapping or transformation of the source data into RDF. However, the data representation required to carry out a knowledge intensive task often emerges from working with data and cannot be wholly specified in advance (this is a crucial requirement of our project SPICE). By distinguishing the processes of re-engineering and re-modelling, Facade-X enables the user to avoid prematurely committing to a mapping and rather focus on querying the data within SPARQL, in a domain-independent way.

`Workflow`. All the technologies considered can in principle be integrated with a typical Semantic Web engineering workflow. However, while we cannot assume that Semantic Web experts have knowledge of RML, XPath, and SPARQL Generate, we can definitely expect knowledge of SPARQL.

`Adaptable`. All technologies provide a flexible set of methods for data manipulation, sparql.aything relying on plain SPARQL. We make the assumption that

---

[19] However, relational tables can be mapped using an approach similar to CSV and spreadsheets tables. A dedicated component is currently being developed.

SPARQL itself is enough for manipulating variables, content types, and RDF structures. It is an interesting, open research question to investigate content manipulation patterns in the various languages and compare their ability to meet user requirements.

`Extendable` and `Sustainable`. Our approach can be implemented within existing SPARQL query processors with minimal development effort. Extending SPARQL Anything requires to write a component that exposes a data source format as Facade-X. Facade-X does not need to be encoded in the software but serves as a reference for mapping an open ended set of formats. In contrast, extending SPARQL Generate and RML requires extending the user toolkit to handle the specificity of the formats, exposing to users new functions for querying, filtering, traversing, and so on. In addition, our approach leads to a more sustainable codebase. To give evidence of this statement, we use the tool cloc[20] to count the lines of Java code required to implement the core module of SPARQL Generate in Apache Jena (without considering format-specific extensions[21]) and the RML implementation in Java[22]. SPARQL Generate and RML require developing and maintaining 12280 and 7951 lines of Java code, respectively. We developed the prototype implementation of SPARQL Anything with 3842 lines of Java code, including all the currently supported transformers.

7 Conclusions

In this paper, we presented an opinionated approach for making non-RDF resources query-able with SPARQL. We contributed a general approach to apply *facades* to content re-engineering and a specific instance of this approach, Facade-X, which defines a general meta-model akin to a *list-of-lists*. We compared our approach with the state of art methods RML and SPARQL Generate and demonstrated how our solution has lower learning demands and cognitive complexity, and it is cheaper to implement and maintain, while having comparable extensibility. Next, we will extend the range of supported formats of SPARQL Anything, including relational databases, Microsoft Office files, and binary content other then images. The approach does not enforce a specific algorithm and leaves open the opportunity of developing alternative strategies for performance optimisation, considering the specificity of the resources, the complexity of the queries, and the computational resources available. Moreover, we will perform a user study for investigating the cognitive implications of using Facade-X as a meta-model with respect to arbitrary RDF, and compare the tools in terms of expressivity and ability to meet user requirements. Finally, other *facades* can be designed as well. It is an interesting research question to investigate content manipulation patterns in alternative facades and evaluate their benefit for content exploration and transformation.

---

[20] cloc: https://github.com/AlDanial/cloc (accessed 15/12/2020).

[21] For SPARQL Generate, we only considered the code included in the submodule sparql-generate-jena.

[22] RMLMapper: https://github.com/RMLio/rmlmapper-java.

References

1. Arenas-Guerrero, J., Scrocca, M., Iglesias-Molina, A., Toledo, J., Pozo-Gilo, L., Dona, D., Corcho, O., Chaves-Fraga, D.: Knowledge graph construction: An etl system-based overview. ESWC 2021 Workshop KGCW (Submission) (2021)
2. Auer, S., Dietzold, S., Lehmann, J., Hellmann, S., Aumueller, D.: Triplify: lightweight linked data publication from relational databases. In: Proceedings of the 18th international conference on World wide web. pp. 621–630 (2009)
3. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C (Feb 2014), https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/
4. Daga, E., d'Aquin, M., Adamou, A., Brown, S.: The open university linked data– data. open. ac. uk. Semantic Web **7**(2), 183–191 (2016)
5. Daga, E., Meroño-Peñuela, A., Motta, E.: Modelling and querying lists in RDF. A pragmatic study. In: Proceedings of the QuWeDa 2019: 3rd Workshop on Querying and Benchmarking the Web of Data co-located with 18th International Semantic Web Conference (ISWC 2019), Auckland, New Zealand, October 26-30, 2019. CEUR-WS.org (2019)
6. Daga, E., Meroño-Peñuela, A., Motta, E.: Sequential linked data: the state of affairs. Semantic Web (2021)
7. Daga, E., Panziera, L., Pedrinaci, C.: A basilar approach for building web apis on top of sparql endpoints. In: CEUR Workshop Proceedings. vol. 1359, pp. 22–32 (2015)
8. Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., Van de Walle, R.: Rml: a generic language for integrated rdf mappings of heterogeneous data. In: 7th Workshop on Linked Data on the Web (2014)
9. Fiorelli, M., Stellato, A.: Lifting tabular data to rdf: A survey. Metadata and Semantic Research **1355**, 85 (2021)
10. Halford, G.S., Andrews, G.: : The development of deductive reasoning: How important is complexity? Thinking & Reasoning **10**(2), 123–145 (2004)
11. Hall, A.G.: The Lish: A Data Model for Grid Free Spreadsheets. Ph.D. thesis, The Open University (2019)
12. Haslhofer, B., Isaac, A.: data. europeana. eu: The europeana linked open data pilot. In: International Conference on Dublin Core and Metadata Applications. pp. 94–104 (2011)
13. Heyvaert, P., De Meester, B., Dimou, A., Verborgh, R.: Declarative rules for linked data generation at your fingertips! In: European Semantic Web Conference. pp. 213–217. Springer (2018)
14. Johnson, R., Vlissides, J.: Design patterns. Elements of Reusable Object-Oriented Software Addison-Wesley, Reading (1995)
15. Lefrançois, M., Zimmermann, A., Bakerally, N.: A sparql extension for generating rdf from heterogeneous formats. In: European Semantic Web Conference. pp. 35–50. Springer (2017)

16. Lieberman, H., Paternò, F., Klann, M., Wulf, V.: End-user development: An emerging paradigm. In: End user development, pp. 1–8. Springer (2006)
17. Meroño-Peñuela, A., Hoekstra, R.: grlc makes github taste like linked data apis. In: European Semantic Web Conference. pp. 342–353. Springer (2016)
18. Michel, F., Faron-Zucker, C., Corby, O., Gandon, F.: Enabling automatic discovery and querying of web apis at web scale using linked data standards. In: Companion Proceedings of The 2019 World Wide Web Conference. pp. 883–892 (2019)
19. Panko, R.R., Aurigemma, S.: Revising the panko–halverson taxonomy of spreadsheet errors. Decision Support Systems **49**(2), 235–244 (2010)
20. Prud'hommeaux, E., Arenas, M., Bertails, A., Sequeda, J.: A direct mapping of relational data to RDF. W3C recommendation, W3C (Sep 2012), https://www.w3.org/TR/2012/REC-rdb-direct-mapping-20120927/
21. Villazón-Terrazas, B.C., Suárez-Figueroa, M., Gómez-Pérez, A.: A pattern-based method for re-engineering non-ontological resources into ontologies. International Journal on Semantic Web and Information Systems (IJSWIS) **6**(4), 27–63 (2010)
22. Warren, P., Mulholland, P.: Using sparql–the practitioners' viewpoint. In: European Knowledge Acquisition Workshop. pp. 485–500. Springer (2018)
23. Warren, P., Mulholland, P., Collins, T., Motta, E.: Making sense of description logics. In: Proceedings of the 11th International Conference on Semantic Systems. pp. 49–56 (2015)
24. Xiao, G., Calvanese, D., Kontchakov, R., Lembo, D., Poggi, A., Rosati, R., Zakharyaschev, M.: Ontology-based data access: A survey. International Joint Conferences on Artificial Intelligence (2018)